Informatics I – HS18

Exercise 7

Submission Details

- **Submission Format:** ZIP file containing your .py solution files
- **Submission Deadline:** 12:00, Tuesday 13th November
- The name of the ZIP file must have the following format: firstname_lastname_studentidentificationnumber_info1_exercise_7.zip, e.g. *hans_muster_12345678_info1_exercise_7.zip*.
- Your ZIP file should contain the following files: `task_1_test.py` and `task_2.py`. **Do not rename these files.**
- **Important:** Please follow the naming conventions very strictly, otherwise we will not be able to grade your submission.
- Please submit the assignment using OLAT. You may upload multiple solutions, but note that **only the last submission** will be evaluated.
- Your submissions will be tested on systems running **Python 3.7.X**. Make sure that your solutions work on this version.

1 Task: Password Manager 5 Points)

In this task, you will create small application that helps you manage all your passwords. A basic description of the functionality is provided here; you will find more detailed instructions below: Your password manager will be initialized with a master password. It will be able to store new passwords, update or retrieve existing ones and list all currently stored passwords. All these functionalities will only be available after the password manager has been unlocked with the correct master password.

Before you start coding, also read the detailed instructions carefully. It is important that you follow the instructions precisely and name your classes, variables and methods exactly as indicated.

Instructions:

- Create a class `Password`:
  - It must have a constructor with the signature `__init__(self, name, username, password)`. Create three private instance variables `__name`, `__username` and `__password` and initialize them in the constructor.
  - Create a method `pretty_str_password()` that returns a string representation of the password object in the following format: *name: username / password* , e.g. `Facebook: JohnDoe1 / foo123`.
  - Also define the `__str__` function to return also a string representation of the password object, but with the masked password (characters replaced with asterisks (*)). The format should be: *name: username / *****, e.g. `Facebook: JohnDoe1 / ******`. The number of asterisks must correspond to the number of characters of the password.
- Create a class `PasswordManager`:
  - It must have a constructor with the signature `__init__(self, master_password)`. Create a private instance variable `__master_password` and initialize it in the constructor.
  - In the constructor, also create a private instance variable `__passwords` which will be a dictionary containing the passwords the manager holds. Finally, create an instance variable `unlocked` which is initially False.
  - Create the functions `lock()` and `unlock(master_password)`. `lock()` should lock the password manager (i.e., set the `unlocked` variable to False) and should not return anything, as it will always succeed. `unlock(master_password)` should unlock the password manager (i.e., set the `unlocked` variable to True) only if the specified master password is correct. It should return a Boolean value indicating success/failure.
  - Create a function `create_new_password(name, username, password)` which checks if a password with the given name already exists and, if not, creates it and stores it in the `__passwords` dictionary (specify the name as key and the `Password` object as value). The function should return the newly created password object or None, if the creation was not successful.
  - Create a function `update_password(name, username, password)` which updates an existing password with a given name, replacing username and password with new values. If the password exists, the function should return the updated password, else it should return None.
  - Create a function `get_password(name)` which returns the `Password` object for a given name, if it exists, and None otherwise.

- **Note:** The functions `create_new_password`, `update_password` and `get_password` should obviously only work if the manager is currently unlocked, else they should always return None.
- Finally, create a function `list_passwords()` which returns a list of string representations of all stored passwords. If the manager is unlocked, the list should contain the `pretty_str_password` representations of the passwords, otherwise the masked `__str__` ones (with the asterisks).

Task: Create a file `task_1.py` and implement the two classes, following the instructions above.

```python
if __name__ == '__main__':
    manager = PasswordManager("abc") # Create a new password manager
    manager.unlock("abc") # Unlock it with the master password

    # Create new passwords
    manager.create_new_password("pw1", "user1", "aaa")
    manager.create_new_password("pw2", "user2", "bbb")

    print(manager.list_passwords()) # Prints the passwords in plain text

    manager.lock() # Lock the password manager

    print(manager.list_passwords()) # Prints the masked passwords
```

Listing 1: Example usage.

2 Task: Spotify Player (5 Points)

In this task, you will emulate the functionality provided by Spotify. *"Spotify is a cloud music streaming app that allows registered users to listen music"*[1]. A basic description of the functionality is provided here; you will find more detailed instructions below: You will have to implement the logic behind playing, pausing and skipping songs.

Before you start coding, also read the detailed instructions carefully. It is important that you follow the instructions precisely and name your classes, variables and methods exactly as indicated.

Instructions:

1. Create a class `Settings`:
   - It must have a constructor with the signature `__init__(self, shuffle, repeat)`. Store the arguments as public instance variables.
     - `repeat` will tell the class `Spotify` whether it should play the playlist on repeat. If yes, the songs in the playlist will loop indefinitely.
     - `shuffle` will tell the class `Spotify` whether it should shuffle the playlist or not. If yes, the songs in the playlist will be picked at random and they will continue playing indefinitely, i.e. similarly as `repeat`.
2. Create a class `Song`:
   - It must have a constructor with the signature `__init__(self, title, artist, duration)`. Store the arguments as public instance variables.
   - Implement the `__eq__` method: it should check for equality based on `title`, `artist` and `duration`.
   - Implement the `__str__` method: the output string should be in the following format: `title - artist: duration`. E.g. `Hotel California - Eagles: 390s`
3. Create a class `Playlist`:
   - It must have a constructor with the signature `__init__(self, title, songs)`. `songs` is a list of `Song` objects. Store the arguments as private instance variables.
   - Implement the `get_title(self)` method: it should return the title of the playlist.
   - Implement the `get_song_titles(self)` method: it should return a list containing the titles of the songs in the playlist, in the same order as they are in `songs`.
   - Implement the `load_song_by_title(self, title)` method: it should search and return the first song with the given title, or `None` if no such song was found in the playlist.
   - Implement the `load_next_song(self, shuffle, repeat` method: it returns the next song based on shuffle and repeat.
     - If `shuffle == True` then it should return any random song (check python's built-in random module, specifically, `randint` and `choice`).
     - Else, if `repeat == True` it should get the next song (based on the previously played song). If the last song played was the last of the playlist, it should start back from the first song.
     - Finally, if `shuffle == False and repeat == False` it should return the next song (based on the previously played song), or `None` if the previously played song was the last one in the playlist.

     Hint: you will likely need one, or some additional instance variables to help keep track which song was last played.

[1] https://support.spotify.com/is/using_spotify/the_basics/what-is-spotify/

4. Finally, create a class `Spotify`:
   - It must have a constructor with the signature `__init__(self, playlist, settings)`. Store the arguments as private instance variables. Additionally initialize two private variables `__current_song` and `__is_playing`. These should be `None` and `False` respectively.
   - Implement the `get_current_song` method which returns the current song.
   - Implement the `is_playing` method which returns `True` if any song is playing, `False` otherwise.
   - Implement the `get_playlist_title` method which returns the title of the playlist.
   - Implement the `play(self, title)` method: `title` is optional and defaults to the empty string.
     - If title is given it should set the current song to the one represented by the title and set the `__is_playing` variable to True, otherwise the current song should be `None` and `__is_playing` should be `False`.
     - If no song is currently playing, it should load from the playlist the next song according to its settings (see the playlist method `load_next_song(self, shuffle, repeat`) and set the `__is_playing` variable to True.
     - If a song is paused, it should set the `__is_playing` variable to True
     - If a song is already playing, it should do nothing
   - Implement the `pause` method which should set the playing flag to `False` but do nothing else otherwise.
   - Implement the `next` method which should skip to the next song according to its settings. If there is no next song as defined in `Playlist`, the current song should be set to `None` and the playing instance variable to `False`.

Check the following snippet to see how to use the classes.

```python
if __name__ == '__main__':
    no_repeat_no_shuffle = Settings(False, False)

    songs = [Song("Hotel California", "Eagles", 390),
             Song("Harder Better Faster Stronger", "Daft Punk", 224),
             Song("2112", "Rush", 1233)]
    playlist = Playlist("MyPlaylist", songs)

    player = Spotify(playlist, no_repeat_no_shuffle)

    assert player.get_playlist_title() == "MyPlaylist"

    # Should be first song, playing
    player.play()
    assert player.get_current_song() == songs[0]
    assert player.is_playing()

    # Should not change song or playing
    player.play()
    assert player.get_current_song() == songs[0]
    assert player.is_playing()

    # Should not change song, playing is False
    player.pause()
    assert player.get_current_song() == songs[0]
    assert not player.is_playing()

    # Should not change song, playing back to True
    player.play()
    assert player.get_current_song() == songs[0]
    assert player.is_playing()

    # Should change song, playing True
    player.next()
    assert player.get_current_song() == songs[1]
    assert player.is_playing()

    # Should change song, playing True
    player.next()
    assert player.get_current_song() == songs[2]
    assert player.is_playing()

    # No songs left, song == None and playing False
    player.next()
    assert not player.get_current_song()
    assert not player.is_playing()

    # Load song by title
    player.play("2112")
    assert player.get_current_song() == songs[2]
    assert player.is_playing()

    # Previous song was last in playlist, next should return None, playing
     False
    player.next()
    assert not player.get_current_song()
    assert not player.is_playing()

    # Start playlist
    player.play()
    assert player.get_current_song() == songs[0]
    assert player.is_playing()

    player.next()
    player.next()
    player.next()
    assert not player.get_current_song()
    assert not player.is_playing()

    # When playlist is finished, calling next starts playlist again.
    player.next()
    assert player.get_current_song() == songs[0]
    assert player.is_playing()
```

Listing 2: Example usage.

Task: Create a file `task_2.py` and implement the classes following the instructions above.