Exercise: Classes and Objects

Problems for exercise and homework for the [Python OOP Course @SoftUni](#).

Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/1937.

1. Vet

Create a class called **Vet**. Upon initialization, it should receive a **name** (string). It should also have an **instance attribute** called **animals** (empty list by default). There should also be **2 class attributes**: **animals** (empty list) which will store the total amount of **animals for all vets**; and **space** (5 by default). You should create **3 additional instance methods**:

- **register_animal(animal_name)**
    - o   If there **is space** in the vet clinic, adds the animal to **both animals' lists** and returns a message:
        "{name} registered in the clinic"
    - o   Otherwise, returns **"Not enough space"**
- **unregister_animal(animal_name)**
    - o   If the animal is **in the clinic**, **removes** it from **both animals'** lists and returns **"{animal} unregistered successfully"**
    - o   Otherwise, returns **"{animal} not in the clinic"**
- **info()**
    - o   Returns info about the vet, the number of animals in his list and the free space in the clinic:
        "{vet_name} has {number_animals} animals. {space_left_in_clinic} space left in clinic"

Examples

Test Code	Output

```peter = Vet("Peter")```
```george = Vet("George")```
```print(peter.register_animal("Tom"))```
```print(george.register_animal("Cory"))```
```print(peter.register_animal("Fishy"))```
```print(peter.register_animal("Bobby"))```
```print(george.register_animal("Kay"))```
```print(george.unregister_animal("Cory"))```
```print(peter.register_animal("Silky"))```
```print(peter.unregister_animal("Molly"))```
```print(peter.unregister_animal("Tom"))```
```print(peter.info())```
```print(george.info())```
	```Tom registered in the clinic```
	```Cory registered in the clinic```
	```Fishy registered in the clinic```
	```Bobby registered in the clinic```
	```Kay registered in the clinic```
	```Cory unregistered successfully```
	```Silky registered in the clinic```
	```Molly not in the clinic```
	```Tom unregistered successfully```
	```Peter has 3 animals. 1 space left in clinic```
	```George has 1 animals. 1 space left in clinic```

2. Time

Create a class called **Time**. Upon initialization, it should receive **hours**, **minutes**, and **seconds** (integers). The class should also have **class attributes max_hours** equal to **23**, **max_minutes** equal to **59**, and **max_seconds** equal to **59**. You should also create **3 additional instance methods**:

- **set_time(hours, minutes, seconds)** - updates the time with the new values
- **get_time()** - returns **"{hh}:{mm}:{ss}"**
- **next_second()** - updates the time with one second (use the **class attributes** for validation) and returns the new time (use the **get_time()** method)

Examples

Test Code	Output
`time = Time(9, 30, 59)` `print(time.next_second())`	`09:31:00`
`time = Time(10, 59, 59)` `print(time.next_second())`	`11:00:00`
`time = Time(23, 59, 59)` `print(time.next_second())`	`00:00:00`

3. Account

Create a class called **Account**. Upon initialization, it should receive an **id** (number), a **name** (string), and a **balance** (integer; **optional**; **0** by default). The class should also have **3 additional instance methods**:

- **credit(amount)** - **adds** the **amount** to the balance and **returns** the **new balance**
- **debit(amount)** - if the amount is **less** than or **equal** to the balance, **reduces** the balance by the amount and **returns** the new balance. Otherwise, return **"Amount exceeded balance"**
- **info()** - returns **"User {name} with account {id} has {balance} balance"**

Examples

Test Code	Output

`account = Account(1234, "George", 1000)`
`print(account.credit(500))`
`print(account.debit(1500))`
`print(account.info())`
	`1500`
	`0`
	`User George with account 1234 has 0 balance`

`account = Account(5411256, "Peter")`
`print(account.debit(500))`
`print(account.credit(1000))`
`print(account.debit(500))`
`print(account.info())`
	`Amount exceeded balance`
	`1000`
	`500`
	`User Peter with account 5411256 has 500 balance`

4. Pizza Delivery

Create a class called **PizzaDelivery**. Upon initialization, it should receive a **name** (string), a **price** (float), and **ingredients** (dictionary). The class should also have an instance attribute **ordered** set to **False** by default. You should also create **3 additional instance methods**:

- **add_extra(ingredient: str, quantity: int, price_per_quantity: float):**
  - o If we already **have this ingredient** in our pizza, **increase the ingredient quantity** with the given one and **update the pizza price** by adding the ingredient price for the given quantity
  - o If we **do not have this ingredient** in our pizza, we should **add it** and **update the pizza price**
- **remove_ingredient(ingredient: str, quantity: int, price_per_quantity: float):**
  - o If we **do not have this ingredient** in our pizza, we should **return** the following message **"Wrong ingredient selected! We do not use {ingredient} in {pizza_name}!"**
  - o If we **have the ingredient**, but we try to remove **more than we have available,** we should **return** the following message **"Please check again the desired quantity of {ingredient}!"**
  - o Otherwise, **remove** the given quantity of the ingredient and update the pizza price by removing the ingredient price for the given quantity

- **make_order()**
    - o Set the attribute **ordered** to **True** and **return** the following message **"You've ordered pizza {pizza_name} prepared with {ingredient: quantity} and the price will be {price}lv."**. The ingredients should be separated by a comma and a space **", "**
    - o Keep in mind that once the pizza is ordered, no further changes are allowed. We should return the following message if the customer tries to change it: **"Pizza {name} already prepared, and we can't make any changes!"**

Examples

Test Code

```margarita = PizzaDelivery('Margarita', 11, {'cheese': 2, 'tomatoes': 1})```
 ```margarita.add_extra('mozzarella', 1, 0.5)```
 ```margarita.add_extra('cheese', 1, 1)```
 ```margarita.remove_ingredient('cheese', 1, 1)```
 ```print(margarita.remove_ingredient('bacon', 1, 2.5))```
 ```print(margarita.remove_ingredient('tomatoes', 2, 0.5))```
 ```margarita.remove_ingredient('cheese', 2, 1)```
 ```print(margarita.make_order())```
 ```print(margarita.add_extra('cheese', 1, 1))```

Output

```Wrong ingredient selected! We do not use bacon in Margarita!```
 ```Please check again the desired quantity of tomatoes!```
 ```You've ordered pizza Margarita prepared with cheese: 0, tomatoes: 1, mozzarella: 1```
 ```and the price will be 9.5lv.```
 ```Pizza Margarita already prepared, and we can't make any changes!```

5. To-do List

In this exercise, we are going to create a whole project step-by-step, starting with the project structure:

```
∨ 📁 project
     🐍 __init__.py
     🐍 section.py
     🐍 task.py
```
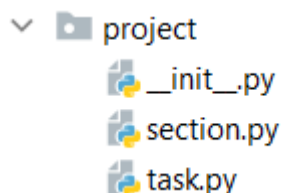
Create separate files for each class, as shown above. You are tasked to create **two classes**: **a Task** class and **a Section** class.

The **Task** class should receive a **name** (string) and a **due_date** (str) upon initialization. A task also has **two attributes**: **comments** (empty list) and **completed** set to **False** by default.

The **Task** class should also have **five additional methods**:

- **change_name(new_name: str)**
    - o Changes **the name of the task** and returns **the new name**.
    - o If the new name **is the same as the current name**, returns **"Name cannot be the same."**
- **change_due_date(new_date: str)**
    - o Changes **the due date of the task** and returns **the new date**.
    - o If the new **date is the same as the current date**, returns **"Date cannot be the same."**
- **add_comment(comment: str)**
    - o Adds a comment to the task.

- **edit_comment(comment_number: int, new_comment: str)**
  - o The **comment number** value represents the **index** of the comment we want to edit. The method should **change the comment** and **return all the comments**, separated **by comma and space (", ")**
  - o If the comment number **is out of range**, returns **"Cannot find comment."**
- **details()**
  - o Returns the task's details in this format:
    > **"Name: {task_name} - Due Date: {due_date}"**

The **Section** class should receive a **name** (string) upon initialization. The task also has **one instance attribute**: **tasks** (empty list)

The Section class should also have **four methods**:

- **add_task(new_task: Task)**
  - o Adds a **new task** to the collection and r**eturns "Task {task details} is added to the section"**
  - o If the task **is already in the collection**, returns **"Task is already in the section {section_name}"**
- **complete_task(task_name: str)**
  - o Changes the task to completed (**True**) and returns **"Completed task {task_name}"**
  - o If the task is not found, returns **"Could not find task with the name {task_name}"**
- **clean_section()**
  - o Removes all the **completed tasks** and returns **"Cleared {amount of removed tasks} tasks."**
- **view_section()**
  - o Returns information about the section and its tasks in this format:
    > **"Section {section_name}:**
    > **{details of the first task}**
    > **{details of the second task}**
    > **…**
    > **{details of the n task}"**

Examples

Test Code	Output
task = Task("Make bed", "27/05/2020") print(task.change_name("Go to University")) print(task.change_due_date("28.05.2020")) task.add_comment("Don't forget laptop") print(task.edit_comment(0, "Don't forget laptop and notebook")) print(task.details()) section = Section("Daily tasks") print(section.add_task(task)) second_task = Task("Make bed", "27/05/2020") section.add_task(second_task) print(section.clean_section()) print(section.view_section())	Go to University 28.05.2020 Don't forget laptop and notebook Name: Go to University - Due Date: 28.05.2020 Task Name: Go to University - Due Date: 28.05.2020 is added to the section Cleared 0 tasks. Section Daily tasks: Name: Go to University - Due Date: 28.05.2020 Name: Make bed - Due Date: 27/05/2020

6. Guild System

You are tasked to create **two classes**: a **Player** class and a **Guild** class.

Follow us:

SoftUni
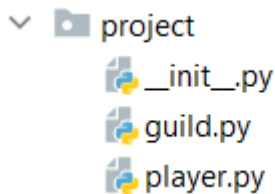
The **Player** class should receive a **name** (string), a **hp** (int), and a **mp** (int) upon initialization. The **Player** also has 2 instance attributes: **skills** (empty dictionary that will contain the skills of each player and its mana cost) and a **guild** set to **"Unaffiliated"** by default.

The Player class should also have **two additional methods**:

- **add_skill(skill_name, mana_cost)**
  - o Adds the skill and the corresponding mana cost to the dictionary of skills. Returns **"Skill {skill_name} added to the collection of the player {player_name}"**
  - o If the skill is already in the collection, returns **"Skill already added"**
- **player_info()**
  - o Returns the player's information, including their skills, in this format:
    ```
    "Name: {player_name}
     Guild: {guild_name}
     HP: {hp}
     MP: {mp}
     ==={skill_name_1} - {skill_mana_cost}
     ==={skill_name_2} - {skill_mana_cost}
     …
     ==={skill_name_N} - {skill_mana_cost}"
    ```

The **Guild** class receives a **name** (string). The **Guild** should also have one instance attribute **players** (an empty list which will contain the players of the guild). The class also has **3 additional methods**:

- **assign_player(player: Player)**
  - o Adds the player to the guild and returns **"Welcome player {player_name} to the guild {guild_name}"**. Remember to change the player's guild in the player class.
  - o If he is already in the guild, returns **"Player {player_name} is already in the guild."**
  - o If the player is in another guild, returns **"Player {player_name} is in another guild."**
- **kick_player(player_name: str)**
  - o Removes the player from the guild and returns **"Player {player_name} has been removed from the guild."**. Remember to change the player's guild in the player class to **"Unaffiliated"**.
  - o If there is no such player in the guild, returns **"Player {player_name} is not in the guild."**
- **guild_info()**
  - o Returns the guild's information, including the players in the guild, in the format:
    ```
    "Guild: {guild_name}
    {first_player's info}
    …
    {Nplayer's info}"
    ```

Examples

Test Code	Output

```
player = Player("George", 50, 100)          Skill Shield Break added to the
print(player.add_skill("Shield Break", 20))  collection of the player George
print(player.player_info())                  Name: George
guild = Guild("UGT")                         Guild: Unaffiliated
print(guild.assign_player(player))           HP: 50
print(guild.guild_info())                    MP: 100
                                             ===Shield Break - 20

                                             Welcome player George to the guild UGT
                                             Guild: UGT
                                             Name: George
                                             Guild: UGT
                                             HP: 50
                                             MP: 100
                                             ===Shield Break - 20
```
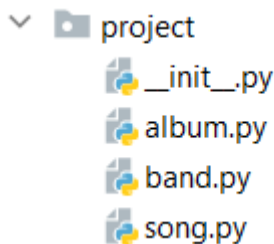
7. Spoopify

You are tasked to create **three classes**: a **Song** class, **an Album** class, and **a Band** class.

```
∨ 📁 project
    📄 __init__.py
    📄 album.py
    📄 band.py
    📄 song.py
```

The **Song** class should receive a **name** (string), a **length** (float), and a **single** (bool) upon initialization. It has **one** method:

- **get_info()**
    - Returns the information of the song in this format: **"{song_name} - {song_length}"**

The **Album** class should receive a **name** (string) upon initialization and might receive **one or more songs**. It also has **instance attributes: published** (**False** by default) and **songs** (an **empty list**). It has **four** methods:

- **add_song(song: Song)**
    - Adds the **song to the album** and returns **"Song {song_name} has been added to the album {name}."**
    - If the song is **single**, returns **"Cannot add {song_name}. It's a single"**
    - If the album is **published**, returns **"Cannot add songs. Album is published."**
    - If the song is **already added**, return **"Song is already in the album."**
- **remove_song(song_name: str)**
    - Removes the song with the given name and returns **"Removed song {song_name} from album {album_name}."**
    - If the song is not in the album, returns **"Song is not in the album."**
    - If the album is published, returns **"Cannot remove songs. Album is published."**

- **publish()**
    - Publishes the album (set to **True**) and returns **"Album {name} has been published."**
    - If the album is published, returns **"Album {name} is already published."**

- **details()**
  - o Returns the information of the album, with the songs in it, in the format:
    ```
    "Album {name}
     == {first_song_info}
     == {second_song_info}
     …
     == {n_song_info}"
    ```

The **Band** class should receive a **name** (string) upon initialization. It also has an **attribute albums** (an **empty list**).

The class has **three** methods:

- **add_album(album: Album)**
  - o Adds an **album to the collection** and returns **"Band {band_name} has added their newest album {album_name}."**
  - o If the album **is already added**, returns **"Band {band_name} already has {album_name} in their library."**
- **remove_album(album_name: str)**
  - o Removes the album from the collection and returns **"Album {name} has been removed."**
  - o If the album is **published**, returns **"Album has been published. It cannot be removed."**
  - o If the album is **not in the collection**, returns **"Album {name} is not found."**
- **details()**
  - o Returns the information of the band, with their albums, in this format:
    ```
    "Band {name}
     {album details}
     ...
     {album details}"
    ```

Examples

Test Code	Output
song = Song("Running in the 90s", 3.45, False) print(song.get_info()) album = Album("Initial D", song) second_song = Song("Around the World", 2.34, False) print(album.add_song(second_song)) print(album.details()) print(album.publish()) band = Band("Manuel") print(band.add_album(album)) print(band.remove_album("Initial D")) print(band.details())	Running in the 90s - 3.45 Song Around the World has been added to the album Initial D. Album Initial D == Running in the 90s - 3.45 == Around the World - 2.34 Album Initial D has been published. Band Manuel has added their newest album Initial D. Album has been published. It cannot be removed. Band Manuel Album Initial D == Running in the 90s - 3.45 == Around the World - 2.34

8. Library*

Create a class called **Library**, where the information regarding the users and books rented/available will be stored. Create another one called **User**, where the information for each of the library users will be stored, and **Registration** class, where user information will be administrated (created/edited/deleted) and stored in the **Library** records.

- project
  - __init__.py
  - library.py
  - registration.py
  - user.py
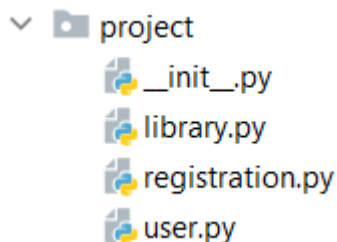
Class User

In the **user.py** file, create class **User**. Upon initialization, it should receive **user_id** (int) and **username** (string). The class should also have an instance attribute **books** that is an empty list. You should also create 2 **instance methods**:

- **info()** - **returns** a string containing the books currently rented by the user in ascending order separated by comma and space**.**
- **__str__()** - override the method to get a string in the following format **"{user_id}, {username}, {list of rented books}"**

Class Library

In the **library.py** create a class **Library**. Upon initialization, it will not receive anything, but it should have the following instance attributes:

- **user_records** - an empty list that will store the users (users objects) of the library
- **books_available** - an empty dictionary that will keep information regarding the authors (key: str) and the books available for each of the authors (list of strings)
- **rented_books** - an empty dictionary that will keep information regarding the **usernames** (key: str) and **nested dictionary** as a value in which will keep information regarding the **book names** (key: str) and **days left** before returning the book to the library (int) - (**{usernames: {book names: days to return}}**).

You should also create **2 additional instance methods**:

- **get_book(author: str, book_name: str, days_to_return: int, user: User):**
  - If the **book is available** in the library adds it to the **books list** for this user, **updates the library records (rented_books and available_books dicts)**, and returns the following message: **"{book_name} successfully rented for the next {days_to_return} days!"**
  - If it is **already rented,** returns the following message **"The book "{book_name}" is already rented and will be available in {days_to_return provided by the user rented the book} days!"**
- **return_book(author:str, book_name:str, user: User):**
  - If the **book is in the user's books list, returns it in the library** (update **books_available and rented_books** class attributes) and **removes it from the books list** for this user
  - Otherwise, **returns** the following message **"{username} doesn't have this book in his/her records!"**

Class Registration

In the **registration.py,** create a class called **Registration**. Upon initialization, It will not receive anything, but we'll have these three methods**.**

- **add_user(user: User, library: Library):**
    - Adds the user if we do not have them in the library's user records already
    - Otherwise, returns the message **"User with id = {user_id} already registered in the library!"**
- **remove_user(user: User, library: Library):**
    - Removes the user from the library records if present
    - Otherwise, returns the message **"We could not find such user to remove!"**
- **change_username(user_id: int, new_username: str, library: Library):**
    - If there is a record with the same user id in the library and the username is different than the provided one, changes the username with the new one provided and returns the message **"Username successfully changed to: {new_username} for user id: {user_id}"**. Changes his username in the **rented_books** dictionary as well (if present).
    - If the new username is the same for this id, returns the following message **"Please check again the provided username - it should be different than the username used so far!"**.
    - If there is no record for the provided id returns **"There is no user with id = {user_id}!"**

Examples

Test Code

```
from project.library import Library
from project.user import User
from project.registration import Registration


user = User(12, 'Peter')
library = Library()
registration = Registration()
registration.add_user(user, library)
print(registration.add_user(user, library))
registration.remove_user(user, library)
print(registration.remove_user(user, library))
registration.add_user(user, library)
print(registration.change_username(2, 'Igor', library))
print(registration.change_username(12, 'Peter', library))
print(registration.change_username(12, 'George', library))

[print(f'{user_record.user_id}, {user_record.username}, {user_record.books}') for
user_record in library.user_records]


library.books_available.update({'J.K.Rowling': ['The Chamber of Secrets',
                                                'The Prisoner of Azkaban',
                                                'The Goblet of Fire',
                                                'The Order of the Phoenix',
                                                'The Half-Blood Prince',
                                                'The Deathly Hallows']})
library.get_book('J.K.Rowling', 'The Deathly Hallows', 17, user)
print(library.books_available)
```

SoftUni

```
print(library.rented_books)
print(user.books)
print(library.get_book('J.K.Rowling', 'The Deathly Hallows', 10, user))
print(library.return_book('J.K.Rowling', 'The Cursed Child', user))
library.return_book('J.K.Rowling', 'The Deathly Hallows', user)
print(library.books_available)
print(library.rented_books)
print(user.books)
```

Output

```
User with id = 12 already registered in the library!
We could not find such user to remove!
There is no user with id = 2!
Please check again the provided username - it should be different than the username
used so far!
Username successfully changed to: George for user id: 12
12, George, []
{'J.K.Rowling': ['The Chamber of Secrets', 'The Prisoner of Azkaban', 'The Goblet of
Fire', 'The Order of the Phoenix', 'The Half-Blood Prince']}
{'George': {'The Deathly Hallows': 17}}
['The Deathly Hallows']
The book "The Deathly Hallows" is already rented and will be available in 17 days!
George doesn't have this book in his/her records!
{'J.K.Rowling': ['The Chamber of Secrets', 'The Prisoner of Azkaban', 'The Goblet of
Fire', 'The Order of the Phoenix', 'The Half-Blood Prince', 'The Deathly Hallows']}
{'George': {}}
[]
```

Test Code

```
from project.library import Library
from project.user import User
from project.registration import Registration


user = User(12, 'Peter')
library = Library()
registration = Registration()
registration.add_user(user, library)
library.books_available.update({'J.K.Rowling': ['The Chamber of Secrets',
                                                'The Prisoner of Azkaban',
                                                'The Goblet of Fire',
                                                'The Order of the Phoenix',
                                                'The Half-Blood Prince',
                                                'The Deathly Hallows']})
library.get_book('J.K.Rowling', 'The Deathly Hallows', 10, user)
print(user)
```

Output

```
12, Peter, ['The Deathly Hallows']
```