# Exercise: Defining Classes

Problems for exercise and homework for the [Python OOP Course @SoftUni](https://softuni.bg/courses/python-oop). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1935>

## Car

Create a class called Car. Upon initialization it should receive a name, model and engine (all strings). Create a method called get\_info() which will return a string in the following format:   
**"**This is {name} {model} with engine {engine}**"**.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| car = Car("Kia", "Rio", "1.3L B3 I4")  print(car.get\_info()) | This is Kia Rio with engine 1.3L B3 I4 |

## Shop

Create a class called Shop. Upon initialization it should receive a name (string) and items (list). Create a method called get\_items\_count() which should return the **amount of items** in the store.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| shop = Shop("My Shop", ["Apples", "Bananas", "Cucumbers"])  print(shop.get\_items\_count()) | 3 |

## Hero

Create a class called Hero. Upon initialization it should receive a name (string) and health (number). Create two functions:

* defend(damage) - Deal the given **damage** to the hero; if the **health** is 0 or less, **set** it **to 0** and **return** **"**{name} was defeated**"**.
* heal(amount) - **Increase the health** of the hero with the given amount.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| hero = Hero("Peter", 100)  print(hero.defend(50))  hero.heal(50)  print(hero.defend(99))  print(hero.defend(1)) | None  None  Peter was defeated |

## Steam User

Create a class called SteamUser. Upon initialization it should receive username (string), games (list). It should also have an **attribute** called played\_hours (**0** by default). Add **three methods** to the class:

* **play(game, hours)**
  + If the **game** is in the user **games increase** the played\_hours by the given hours and return "{username} is playing {game}**"**
  + Otherwise, return **"**{game} is not in library**"**
* **buy\_game(game)**
  + If the game **is not** already in the user's **games**, **add it** and return **"**{username} bought {game}**"**
  + Otherwise return **"**{game} is already in your library**"**
* **stats()** - returns "{username} has {games\_count} games. Total play time: {played\_hours}**"**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| user = SteamUser("Peter", ["Rainbow Six Siege", "CS:GO", "Fortnite"])  print(user.play("Fortnite", 3))  print(user.play("Oxygen Not Included", 5))  print(user.buy\_game("CS:GO"))  print(user.buy\_game("Oxygen Not Included"))  print(user.play("Oxygen Not Included", 6))  print(user.stats()) | Peter is playing Fortnite  Oxygen Not Included is not in library  CS:GO is already in your library  Peter bought Oxygen Not Included  Peter is playing Oxygen Not Included  Peter has 4 games. Total play time: 9 |

## Programmer

Create a class called Programmer. Upon initialization it should receive name (string), language (string), skills (integer). The class should have **two methods**:

* **watch\_course(course\_name, language, skills\_earned)**
  + If the programmer's **language** is the **equal** to the **one on the course increase his skills** with the given one and return a message **"**{programmer} watched {course\_name}**"**.
  + Otherwise return **"**{name} does not know {language}**"**.
* **change\_language(new\_language, skills\_needed)** 
  + If the programmer **has the skills** and the **language is different from his**, **change** his language to the new one and return **"**{name} switched from {previous\_language} to {new\_language}**"**.
  + If the programmer **has the skills**, but the **language is the same** as his return **"**{name} already knows {language}**"**.
  + In the last case the programmer does **not have the skills**, so return **"**{name} needs {needed\_skills} more skills**"** and **don't change his language**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| programmer = Programmer("John", "Java", 50)  print(programmer.watch\_course("Python Masterclass", "Python", 84))  print(programmer.change\_language("Java", 30))  print(programmer.change\_language("Python", 100))  print(programmer.watch\_course("Java: zero to hero", "Java", 50))  print(programmer.change\_language("Python", 100))  print(programmer.watch\_course("Python Masterclass", "Python", 84)) | John does not know Python  John already knows Java  John needs 50 more skills  John watched Java: zero to hero  John switched from Java to Python  John watched Python Masterclass |

***Note: For the rest of the problems please submit a zip file, containing a separate file for each of the classes, with the class names provided in the problem description and include them in a module named project.***

## Pokemon Battle

You are tasked to create **two classes**: **a Pokemon** class and **a Trainer** class. The **Pokemon** class should receive a **name** (string) and **health** (int) upon initialization. It should also have a method called **pokemon\_details** that returns **the information of the pokemon: "{pokemon\_name} with health {pokemon\_health}"**

The **Trainer** class should receive a **name** (string). The Trainer should also have an attribute **pokemon** (list, empty by default). The Trainer has **three methods**:

* **add\_pokemon(pokemon: Pokemon)**
  + Add the **pokemon to the collection**. After adding the pokemon, it **should return** **"Caught {pokemon\_name} with health {pokemon\_health}"**. Note: use the pokemon's details method.
  + If the pokemon is already in the collection, it should return **"This pokemon is already caught"**
* **release\_pokemon(pokemon\_name: String)** 
  + Check if you have a pokemon with the name and **remove it from the collection**. It should return **"You have released {pokemon\_name}"**
  + If there **isn't a pokemon** with that name in the collection, return **"Pokemon is not caught"**
* **trainer\_data()**
  + The method returns the information of the trainer with his pokemon in this format:

**"Pokemon Trainer {trainer\_name}  
 Pokemon count {the amount of pokemon caught}  
 - {pokemon\_details}**

**...  
 - {pokemon\_details}**

**"**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| pokemon = Pokemon("Pikachu", 90)  print(pokemon.pokemon\_details())  trainer = Trainer("Ash")  print(trainer.add\_pokemon(pokemon))  second\_pokemon = Pokemon("Charizard", 110)  print(trainer.add\_pokemon(second\_pokemon))  print(trainer.add\_pokemon(second\_pokemon))  print(trainer.release\_pokemon("Pikachu"))  print(trainer.release\_pokemon("Pikachu"))  print(trainer.trainer\_data()) | Pikachu with health 90  Caught Pikachu with health 90  Caught Charizard with health 110  This pokemon is already caught  You have released Pikachu  Pokemon is not caught  Pokemon Trainer Ash  Pokemon count 1  - Charizard with health 110 |

## Guild System

You are tasked to create **two classes**: **a Player** class and **a Guild** class. The **Player** class should receive a **name** (string), **hp** (int) and **mp** (int) upon initialization. The Player also has 2 attributes: **skills** (empty dictionary) and **guild** set to **"Unaffiliated"** by default.

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

* **add\_skill(skill\_name, mana\_cost)**
  + Add the skill to the collection. Return **"Skill {skill\_name} added to the collection of the player {player\_name}"**
  + If the skill is already in the collection, return **"Skill already added"**
* **player\_info()** 
  + Returns the player's information, including his/her 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 receive a **name** {string}. It creates **a list of players** (empty by initialization). The class also has 3 methods:

* **assign\_player(player: Player)**
  + Add the player to the guild. Return **"Welcome player {player\_name} to the guild {guild\_name}".** Remember to change the player's guild in the player class.
  + If the player is already in the guild, return **"Player {player\_name} is already in the guild."**
  + If the player is in another guild, return **"Player {player\_name} is in another guild."**
* **kick\_player(player\_name: String)**
  + Remove the player to the guild. Return **"Player {player\_name} has been removed from the guild.".** Remember to change the player's guild in the player class.
  + If the isn't a player with that name in the guild, return **"Player {player\_name} is not in the guild."**
* **guild\_info()** 
  + Returns the guild's information, including the players in the guild, in this format:

**"Guild: {guild\_name}  
 {player's info}  
"**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| player = Player("George", 50, 100)  print(player.add\_skill("Shield Break", 20))  print(player.player\_info())  guild = Guild("UGT")  print(guild.assign\_player(player))  print(guild.guild\_info()) | Skill Shield Break added to the collection of the player George  Name: George  Guild: Unaffiliated  HP: 50  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 |

## Spoopify

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

The **Song** class should receive a **name** (string), **length** (float) and **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) and **songs(one, many or none)** **as arguments** upon initialization. It also has an **attribute published** (**False** by default). It has **four** methods:

* **add\_song(song: Song)**
  + Adds the **song to the album**. Return **"Song {song name} has been added to the album {name}."**
  + If the song is **single**, return **"Cannot add {song name}. It's a single"**
  + If the album is **published**, return **"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 return **"Removed song {song name} from album {name}.**
  + If the song is not in the album, return **"Song is not in the album."**
  + If the album is published, return **"Cannot remove songs. Album is published."**
* **publish()**
  + Publish the album and return **"Album {name} has been published."**
  + If the album is published, return **"Album {name} is already published."**
* **details()**
  + Returns the information of the album, with the songs in it, in this 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** (**empty list**). It has **three** method:

* **add\_album(album: Album)**
  + Adds an **album to the collection** and returns **"Band {name} has added their newest album {name}."**
  + If the album **is already added**, return **"Band {name} already has {album name} in their library."**
* **remove\_album(album\_name: str)**
  + Removes the album from the collection and returns **"Album {name} has been removed."**
  + If the album is **published**, return **"Album has been published. It cannot be removed."**
  + If the album is **not in the collection**, return **"Album {name} is not found."**
* **details()**
  + 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 |

## Todo List

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. The Task also has **two attributes**: **comments** (empty list) and **completed** set to **False** by default.

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

* **change\_name(new\_name: str)**
  + Change **the name of the task** and return **the new name**.
  + If the new name **is the same as the current name**, return **"Name cannot be the same."**
* **change\_due\_date(new\_date: str)** 
  + Change **the due date of the task** and return **the new date**.
  + If the new **date is the same as the current date**, return **"Date cannot be the same."**
* **add\_comment(comment: str)**
  + Add a comment to the task.
* **edit\_comment(comment\_number: int, new\_comment: str)**
  + If the **comment\_number is in the comments**, change **the comment** and **return all of the comments**, separated **by comma and space (", ")**
  + If the comment\_number **is not in the comments**, return **"Cannot find comment."**
* **details()**
  + Return 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 attributes**: **tasks** (empty list)

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

* **add\_task(new\_task: Task)**
  + Add a **new task** to the collection. **Return "Task {task details} is added to the section"**
  + If the task **is in the collection**, return **"Task is already in the section {section\_name}"**
* **complete\_task(task\_name: str)** 
  + Change the task to **completed**. Return **"Completed task {task\_name}"**
  + If the task is not found, return **"Could not find task with the name {task\_name}"**
* **clean\_section()**
  + Removes all of the **completed tasks** and returns **"Cleared {amount of removed tasks} tasks."**
* **view\_section()**
  + Return information about the section and it's 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 |