# Exercises: Models Inheritance and Customization

This document defines the **exercise assignments** for the [Python ORM course @ Software University](https://softuni.bg/trainings/4253/python-orm-october-2023).

Submit your solutions in the SoftUni [Judge system](https://judge.softuni.org/Contests/4330/Models-Inheritance-and-Customization-Exercise).

## Character Classes

You are given a task to create and explore character class specializations. Imagine you have various character classes, each with unique attributes and abilities. The goal is to define these character classes and their specializations.

### Model BaseCharacter

The model "**BaseCharacter**" should be implemented. It is a base model and **is NOT meant to create a database table on its own**. The model has the following fields:

* **"name"** - character field, **consisting of a maximum of 100 characters.**
* **"description"** - text field.

### Model Mage

The model "**Mage**" should be implemented. It is a type of **character**. The model has the following fields:

* **"elemental\_power"** - character field, **consisting of a maximum of 100 characters.**
* **"spellbook\_type"** - character field, **consisting of a maximum of 100 characters.**

### Model Assassin

The model "**Assassin**" should be implemented. It is a type of **character**. The model has the following fields:

* **"weapon\_type"** - character field, **consisting of a maximum of 100 characters.**
* **"assassination\_technique"** - character field, **consisting of a maximum of 100 characters.**

### Model DemonHunter

The model "**DemonHunter**" should be implemented. It is a type of **character**. The model has the following fields:

* **"weapon\_type"** - character field, **consisting of a maximum of 100 characters.**
* **"demon\_slaying\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model TimeMage

The model "**TimeMage**" should be implemented. It is a type of **mage**. The model has the following fields:

* **"time\_magic\_mastery"** - character field, **consisting of a maximum of 100 characters.**
* **"temporal\_shift\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model Necromancer

The model "**Necromancer**" should be implemented. It is a type of **mage**. The model has the following fields:

* **"raise\_dead\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model ViperAssassin

The model "**ViperAssassin**" should be implemented. It is a type of **assassin**. The model has the following fields:

* **"venomous\_strikes\_mastery"** - character field, **consisting of a maximum of 100 characters.**
* **"venomous\_bite\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model ShadowbladeAssassin

The model "**ShadowbladeAssassin**" should be implemented. It is a type of **assassin**. The model has the following fields:

* **"shadowstep\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model VengeanceDemonHunter

The model "**VengeanceDemonHunter**" should be implemented. It is a type of **demon hunter**. The model has the following fields:

* **"vengeance\_mastery"** - character field, **consisting of a maximum of 100 characters.**
* **"retribution\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Model FelbladeDemonHunter

The model "**FelbladeDemonHunter**" should be implemented. It is a type of **demon hunter**. The model has the following fields:

* **"felblade\_ability"** - character field, **consisting of a maximum of 100 characters.**

### Examples

**When submitting your solution to the Judge system, please, refactor the caller.py file as you comment or delete the creation of the objects, otherwise, it will have an impact on the database and the results of the Judge tests.**

|  |
| --- |
| **Test Code - caller.py** |
| # Create instances mage = Mage.objects.create(  name="Fire Mage",  description="A powerful mage specializing in fire magic.",  elemental\_power="Fire",  spellbook\_type="Ancient Grimoire" )  necromancer = Necromancer.objects.create(  name="Dark Necromancer",  description="A mage specializing in dark necromancy.",  elemental\_power="Darkness", spellbook\_type="Necronomicon",  raise\_dead\_ability="Raise Undead Army" )  print(mage.elemental\_power) print(mage.spellbook\_type) print(necromancer.name)  print(necromancer.description) print(necromancer.raise\_dead\_ability) |
| **Output** |
| Fire  Ancient Grimoire  Dark Necromancer  A mage specializing in dark necromancy.  Raise Undead Army |

## Chat App

Currently, you are building a basic **messaging** **system** for a social networking platform. Users can send messages to each other, mark messages as **read** or **unread**, **reply** to messages, and **forward** messages to other users.

### Model UserProfile

Create a new Django model "**UserProfile**" with the provided information:

* "**username**" - character field, **consisting of a maximum of 70 characters, unique.**
* **"email**" - email field, **unique.**
* **"bio**" - text field, **optional.**

### Model Message

Create a new Django model "**Message**" with the provided information:

* "**sender**" - many-to-one relation, **with related name "sent\_messages".** If a sender is **deleted**, you should **automatically** **delete** all the **related** messages.
* **"receiver**" - many-to-one relation, **with related name "received\_messages".** If a receiver is **deleted**, you should **automatically** **delete** all the **related** messages.
* **"content**" - text field**.**
* **"timestamp**" - date time field**. When a record is created you should save the time of the creation.**
* **"is\_read**" - boolean field, with **default** value "**False**"**.**

### Methods inside the Message model

Method**:** **"mark\_as\_read()"** **mark** the **message as read**.

Method**:** **"mark\_as\_unread()"** **marks** the **message as unread**.

Method**: reply\_to\_message(reply\_content, receiver)"** **replies** to messages. **Create** a new **message** with a new **sender**, a new **receiver**, and new **reply** **content,** **save** it in the **database**, and **return** the **message** object.

Method**: "forward\_message(sender, receiver)"** **forwards** messages. **Create** a new message with a new sender, a new **receiver**, and the **content** from the **message** to be **forwarded**, **save** it in the **database**, and **return** the **message** object.

### Examples

**When submitting your solution to the Judge system, please, refactor the caller.py file as you comment or delete the creation of the objects, otherwise, it will have an impact on the database and the results of the Judge tests.**

|  |
| --- |
| **Test Code - caller.py** |
| # Create users user1 = UserProfile.objects.create(username='john\_doe', email='john@example.com', bio='Hello, I am John Doe.')  user2 = UserProfile.objects.create(username='jane\_smith', email='jane@example.com', bio='Hi there, I am Jane Smith.')  user3 = UserProfile.objects.create(username='alice', email='alice@example.com', bio='Hello, I am Alice.')  # Create a message from user1 to user2 message1 = Message.objects.create(  sender=user1,  receiver=user2,  content="Hello, Jane! Could you please tell Alice that tomorrow we are going on vacation?") print(message1.content)  # Mark the message as read message1.mark\_as\_read() print(f"Is read: {message1.is\_read}")  # Create a reply from user2 to user1 reply\_message = message1.reply\_to\_message(  receiver=user1,  reply\_content="Hi John, sure! I will forward this message to her!") print(reply\_message.content)  # Create a forwarded message from user2 to user3 forwarded\_message = message1.forward\_message(sender=user2, receiver=user3) print(f"Forwarded message from {forwarded\_message.sender.username} to {forwarded\_message.receiver.username}") |
| **Output** |
| Hello, Jane! Could you please tell Alice that tomorrow we are going on vacation?  Is read: True  Hi John, sure! I will forward this message to her!  Forwarded message from jane\_smith to alice |

## Student Information

Write a Django model "Student" with the provided information:

* **"name**" - character field**, consisting of a maximum of 100 characters.**
* **"student\_id**" - custom "**StudentIDField"** field.

### Field StudentIDField

**In the "main\_app", the field** "**StudentIDField"** is a **type** of **positive integer** and **returns** information about the **student** **id**. It should **save** the **id** for every **student** in the database as a **positive** **integer**. When creating an **instance**, you can pass as arguments **floats**, and even **strings (string numbers)**.

### Examples

**When submitting your solution to the Judge system, please, refactor the caller.py file as you comment or delete the creation of the objects, otherwise, it will have an impact on the database and the results of the Judge tests.**

|  |
| --- |
| **Test Code - caller.py** |
| # Test cases student1 = Student(name="John", student\_id=12345) student1.save()  student2 = Student(name="Alice", student\_id=45.23) student2.save()  student3 = Student(name="Bob", student\_id="789") student3.save()  # Retrieving student IDs from the database retrieved\_student1 = Student.objects.get(name="John") retrieved\_student2 = Student.objects.get(name="Alice") retrieved\_student3 = Student.objects.get(name="Bob")  print(retrieved\_student1.student\_id) print(retrieved\_student2.student\_id) print(retrieved\_student3.student\_id) |
| **Output** |
| 12345  45  789 |

## Credit Card Masking

Write a Django model "**CreditCard**" with the provided information:

* **"card\_owner**" - character field**, consisting of a maximum of 100 characters.**
* **"card\_number**" - custom "**MaskedCreditCardField"** field. **Initialize** a "**max\_length"** of **20**.

### Field MaskedCreditCardField

In the **"main\_app"**, **the field** "**MaskedCreditCardField"** is a **type** of **character field** and **returns** information about the credit **card** **number**. It should **save** the **card number in a masked format in the database** as a string with only the **card's last four digits** visible in the format: **"\*\*\*\*-\*\*\*\*-\*\*\*\*-{last\_four\_card\_digits}**"

* If a **data type** other than a **string** is provided as the **card number**, a "**ValidationError"** should be **raised** with the message: **"The card number must be a string"**.
* The **card number** can consist only of **digits**, otherwise a "**ValidationError"** should be **raised** with the message: **"The card number must contain only digits"**.
* The **card number** must be exactly **16** **digits long**, otherwise a "**ValidationError"** should be **raised** with the message: **"The card number must be exactly 16 characters long"**.

### Examples

**When submitting your solution to the Judge system, please, refactor the caller.py file as you comment or delete the creation of the objects, otherwise, it will have an impact on the database and the results of the Judge tests.**

|  |
| --- |
| **Test Code - caller.py** |
| # Create CreditCard instances with card owner names and card numbers credit\_card1 = CreditCard.objects.create(card\_owner="Krasimir", card\_number="1234567890123450") credit\_card2 = CreditCard.objects.create(card\_owner="Pesho", card\_number="9876543210987654") credit\_card3 = CreditCard.objects.create(card\_owner="Vankata", card\_number="4567890123456789")  # Save the instances to the database credit\_card1.save() credit\_card2.save() credit\_card3.save()  # Retrieve the CreditCard instances from the database credit\_cards = CreditCard.objects.all()  # Display the card owner names and masked card numbers for credit\_card in credit\_cards:  print(f"Card Owner: {credit\_card.card\_owner}")  print(f"Card Number: {credit\_card.card\_number}") |
| **Output** |
| Card Owner: Krasimir  Card Number: \*\*\*\*-\*\*\*\*-\*\*\*\*-3450  Card Owner: Pesho  Card Number: \*\*\*\*-\*\*\*\*-\*\*\*\*-7654  Card Owner: Vankata  Card Number: \*\*\*\*-\*\*\*\*-\*\*\*\*-6789 |

## \*Hotel Reservation System

You've been tasked with creating a cutting-edge reservation system for top-tier hotels. This hotel wants to step up its game by introducing personalized pricing and handling special guest requests. Your mission is to extend this reservation system to handle **'Special Reservations**' and **'Extended Reservations**'. Special Reservations should be able to accommodate unique guest requests, and Extended Reservations should allow guests to extend their stays.

### Model Hotel

The model "**Hotel**" should be implemented. The **model** has the following **fields**:

* **"name"** - character field**, consisting of a maximum of 100 characters.**
* **"address"** - character field**, consisting of a maximum of 200 characters.**

### Model Room

The model "**Room**" should be implemented. The **model** has the following **fields**:

* "**hotel**" - many-to-one relation to the "**Hotel"** model. If a hotel is **deleted**, you should **automatically** **delete** all the **related** rooms.
* **"number"** - character field**, consisting of a maximum of 100 characters, unique.**
* **"capacity"** - positive integer field.
* **"total\_guests"** - positive integer field.
* **"price\_per\_night"** - decimal field, **consisting of a maximum of 10 digits and 2 decimal places.**

### Methods inside the Room model

Before **saving** an **instance** of type **room in the database**:

* If the **total** **number** of **guests** is **greater** than the **capacity** of the room, a "**ValidationError**" should be **raised** with the **message** - **"Total guests are more than the capacity of the room"**.
* If the room is **saved** successfully, **return** the message: **"Room {room\_number} created successfully"**.

### Model BaseReservation

The model "**BaseReservation**" should be implemented. It is a **base** model and **is NOT meant to create a database table on its own**. The model has the following fields:

* **"room"** - many-to-one relation to the **"Room"** class. If a **room** is **deleted**, you should **automatically** **delete** all the **related** reservations.
* **"start\_date"** - date field.
* **"end\_date"** - date field.

### Methods inside the BaseReservation model

Method**:** **"reservation\_period()"** **returns** the **reservation period** in **days** (**integer)**.

Method**:** **"calculate\_total\_cost()"** **returns** the total cost as you multiply the **price** **per** **night** by the **reservation** **period** (**in** **days), formatted** to the second **decimal place**.

### Model RegularReservation

The model "**RegularReservation**" should be implemented. It is a model of **type** **reservation**.

### Methods inside the RegularRegistration model

Before **saving** an **instance** of type **regular** **reservation in the database**, check if the **reservation** **dates** are implemented correctly:

* If the **start date** is **greater than or equal** to the **end date**, a "**ValidationError**" should be **raised** with the **message** - "**Start date cannot be after or in the same end date**".
* If the **reservation** being created overlaps with **existing** **reservations** (i.e., it has **dates** that match other **reservations**), a "**ValidationError"** should be **raised** with the message - **"Room {room\_number} cannot be reserved".** A **conflicting reservation** occurs when the **date** range specified for the new **reservation** clashes with the **dates** of **reservations** that already **exist**.
* If the **registration** is **saved** successfully, **return** the message: **"Regular reservation for room {room\_number}"**.

**Please be aware that all types of reservations are intended to span until the end date, including the end date itself.**

### Model SpecialReservation

The model "**SpecialReservation**" should be implemented. It is a model of **type** **reservation**.

### Methods inside the SpecialRegistration model

Before **saving** an **instance** of type **special** **reservation in the database**, check if the **reservation** **dates** are implemented correctly:

* If the **start date** is **greater than or equal** to the **end date**, a "**ValidationError**" should be **raised** with the **message** - "**Start date cannot be after or in the same end date**".
* If the **reservation** being created overlaps with **existing** **reservations** (i.e., it has **dates** that match other **reservations**), a "**ValidationError"** should be **raised** with the message - **"Room {room\_number} cannot be reserved".** A **conflicting reservation** occurs when the **date** range specified for the new **reservation** clashes with the **dates** of **reservations** that already **exist**.
* If the **registration** is **saved** successfully, **return** the message: **"Special reservation for room {room\_number}"**.

**Please be aware that all types of reservations are intended to span until the end date, including the end date itself.**

Method**:** **"extend\_reservation(days: int)"** **extends** existing **reservations** with the given **days**.

* You should **extend** an already existing **reservation**. If the room is not reserved or you try to **extend** the **reservation** period and the **room** has been already **reserved** for the desired **period**, a **"ValidationError"** shouldbe **raised** with the message - **"Error during extending reservation"**.
* If the **extending** is **successful**, you should **return** the **message**: "**Extended reservation for room {room\_number} with {days} days".**

|  |
| --- |
| **Test Code - caller.py** |
| # Create a Hotel instance hotel = Hotel.objects.create(name="Hotel ABC", address="123 Main St")  # Create Room instances associated with the hotel room1 = Room.objects.create(  hotel=hotel,  number="101",  capacity=2,  total\_guests=1,  price\_per\_night=100.00 )  # Create SpecialReservation instances special\_reservation1 = SpecialReservation(  room=room1,  start\_date=date(2023, 1, 1),  end\_date=date(2023, 1, 5) )  print(special\_reservation1.save())  special\_reservation2 = SpecialReservation(  room=room1,  start\_date=date(2023, 1, 10),  end\_date=date(2023, 1, 12) )  print(special\_reservation2.save())  print(special\_reservation1.calculate\_total\_cost()) print(special\_reservation1.reservation\_period())  # Example of extending a SpecialReservation try:  special\_reservation1.extend\_reservation(5) except ValidationError as e:  print(e) |
| **Output** |
| Special reservation for room 101  Special reservation for room 101  ['Error during extending reservation']  400.00  4 |