# Exercises: Data Operations in Django with Queries

This document defines the **exercise assignments** for the [**Python ORM course @ Software University**](https://softuni.bg/modules/137/python-db).

Submit your solutions in the SoftUni [**Judge system**](https://judge.softuni.org/Contests/4324/Data-Operations-in-Django-with-Queries-Exercise).

## Pet

Write a Django model called **"Pet"** with the provided **fields**:

* **name** - character field, **consisting of a maximum of 40 characters.**
* **species** - character field, **consisting of a maximum of 40 characters.**

### Functions inside the caller.py file

**create\_pet(name: str, species: str)** **creates** a pet, **saves** it in the database, and **returns** its **name** and  **species** as a string as follows:

* **"{pet\_name} is a very cute {pet\_species}!"**

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |
| --- | --- |
| **name** | **species** |
| Buddy | Dog |
| Whiskers | Cat |
| Rocky | Hamster |

**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** |
| print(create\_pet('Buddy', 'Dog')) print(create\_pet('Whiskers', 'Cat')) print(create\_pet('Rocky', 'Hamster')) |
| **Output** |
| Buddy is a very cute Dog!  Whiskers is a very cute Cat!  Rocky is a very cute Hamster! |

## Artifact

Write a Django model called **"Artifact"** with the provided **fields**:

* **name** - character field, **consisting of a maximum of 70 characters.**
* **origin** - character field, **consisting of a maximum of 70 characters.**
* **age** - positive integer field**.**
* **description** - text field**.**
* **is\_magical** - boolean field with **default** value **"False"**.

### Functions inside the caller.py file

**create\_artifact(name: str, origin: str, age: int, description: str, is\_magical: bool)** **creates** an artifact, **saves** it in the database, and **returns** its **name** and **age** as a string as follows:

* **"The artifact {name} is {age} years old!"**

**rename\_artifact(artifact: Artifact, new\_name: str)** **renames** the given artifact, only if it is **magical** and the **age** is greater than **250** **years** then **saves** the changes in the database.

**delete\_all\_artifacts()** **deletes** all **artifacts** from the database.

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **name** | **origin** | **age** | **description** | **is\_magical** |
| Ancient Sword | Lost Kingdom | 500 | A legendary sword with a rich history | True |
| Crystal Amulet | Mystic Forest | 300 | A magical amulet believed to bring good fortune | True |
| Stone Tablet | Ruined Temple | 1000 | An ancient tablet covered in mysterious inscriptions | False |

**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** |
| print(create\_artifact('Ancient Sword', 'Lost Kingdom', 500, 'A legendary sword with a rich history', True))  artifact\_object = Artifact.objects.get(name='Ancient Sword')  rename\_artifact(artifact\_object, 'Ancient Shield')  print(artifact\_object.name) |
| **Output** |
| The artifact Ancient Sword is 500 years old!  Ancient Shield |

## Location

Write a Django model called **"Location"** with the provided **fields**:

* **name** - character field, **consisting of a maximum of 100 characters.**
* **region** - character field, **consisting of a maximum of 50 characters.**
* **population** - positive integer field.
* **description** - text field**.**
* **is\_capital** - boolean field with **default** value **"False"**.

### Functions inside the caller.py file

**show\_all\_locations()** **returns** the **name** and the **population** for every location, **ordered** by **id** (**descending**) as a string as follows:

* **"{name\_1} has a population of {population\_1}!**

**…**

**{name\_N} has a population of {population\_N}!"**

**new\_capital()** makes the **first** **location** **capital**. **Do not forget to save the changes.**

**get\_capitals()** **returns** the locations which are **capitals** (**as a queryset with data only for the name of the location**)**.**

**delete\_first\_location()** **deletes** the **first** **location** from the database.

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **name** | **region** | **population** | **description** | **is\_capital** |
| Sofia | Sofia Region | 1329000 | The capital of Bulgaria and the largest city in the country | False |
| Plovdiv | Plovdiv Region | 346942 | The second-largest city in Bulgaria with a rich historical heritage | False |
| Varna | Varna Region | 330486 | A city known for its sea breeze and beautiful beaches on the Black Sea | False |

**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** |
| print(show\_all\_locations())  print(new\_capital())  print(get\_capitals()) |
| **Output** |
| Varna has a population of 330486!  Plovdiv has a population of 346942!  Sofia has a population of 1329000!  None  <QuerySet [{'name': 'Sofia'}]> |

### Hint

* You can **return** the **queryset** as required with the method "**values()**"

## Car

Write a Django model called **"Car"** with the provided **fields**:

* **model** - character field, **consisting of a maximum of 40 characters.**
* **year** - positive integer field.
* **color** - character field, **consisting of a maximum of 40 characters.**
* **price** - decimal field, with **a maximum of 10 digits** and **2 decimal places.**
* **price\_with\_discount** - decimal field, with a **maximum of 10 digits** and **2 decimal places**, and a **default** value of **0** (**zero**).

### Functions inside the caller.py file

**apply\_discount()** modifies the **price** **with** a **discount** field.

* For every car must be applied a **discount** with the sum of the **digits** of the **year** as **a percentage** (e.g. if the **year** is **2014** the percentage **discount** is **7%**). The newly generated **price** must be **saved** in the **price with a discount field.** **Do not modify the original price field.**

**get\_recent\_cars()** **returns** all **cars** manufactured since the **year** **2020** (**exclusive**)**,** (**as a queryset with data for the model and the price with a discount for the recent cars**).

**delete\_last\_car()** **deletes** the **last** **car** from the database.

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |  |  |
| --- | --- | --- | --- |
| **model** | **year** | **color** | **price** |
| Mercedes C63 AMG | 2019 | white | 120000.00 |
| Audi Q7 S line | 2023 | black | 183900.00 |
| Chevrolet Corvette | 2021 | dark grey | 199999.00 |

**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** |
| apply\_discount() print(get\_recent\_cars()) |
| **Output** |
| <QuerySet [{'model': 'Audi Q7 S line', 'price\_with\_discount': Decimal('171027.00')}, {'model': 'Chevrolet Corvette', 'price\_with\_discount': Decimal('189999.05')}]> |

## Task Encoder

Write a Django model called **"Task"** with the provided **fields**:

* **title** - character field, **consisting of a maximum of 25 characters.**
* **description** - text field.
* **due\_date** - date field.
* **is\_finished** - boolean field with **default** value **"False"**.

### Functions inside the caller.py file

**show\_unfinished\_tasks()** **returns** all **incomplete** **tasks** with their **title** and **due** **date** as a string as follows:

* **"Task - {title\_1} needs to be done until {due\_date\_1}!**

**…**

**Task - {title\_N} needs to be done until {due\_date\_N}!"**

**complete\_odd\_tasks()** makes every **task** withanodd **id** **finished**.

**encode\_and\_replace(text: str**, **task\_title: str)** **encodes** the text and **replaces** it with the **description** for all tasks with the given **title**. The **encoded** **text** should be **3** **ASCII** symbols below the given one.

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |  |  |
| --- | --- | --- | --- |
| **title** | **description** | **due\_date** | **is\_finished** |
| Sample Task | This is a sample task description | 2023-10-31 | False |

**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** |
| encode\_and\_replace("Zdvk#wkh#glvkhv$", "Simple Task")  print(Task.objects.get(title='Simple Task').description) |
| **Output** |
| Wash the dishes! |

## Hotel Room

Write a Django model called **"HotelRoom"** with the provided **fields**:

* **room\_number** - positive integer field.
* **room\_type** - character field, **consisting of a maximum of 10 characters** with **choices** - **"Standard"**, **"Deluxe"**, and **"Suite"**.
* **capacity** - positive integer field.
* **amenities** - text field.
* **price\_per\_night** - decimal field with a **maximum** of 8 **digits** and 2 **decimal** **places**.
* **is\_reserved** - boolean field with **default** value **"False"**.

### Functions inside the caller.py file

**get\_deluxe\_rooms()** **returns all deluxe rooms** with their **room** **number** and **price** **per night**,only if their **id** is **even** as a string as follows:

* **"Deluxe room with number {room\_number\_1} costs {price\_per\_night\_1}$ per night!**

**…**

**Deluxe room with number {room\_number\_N} costs {price\_per\_night\_N}$ per night!"**

**increase\_room\_capacity()** increasesthe **capacity** ofevery **reserved room** with the **capacity** of the **previous room** inthe database, **ordered** by **id** (**ascending**).

* If there is only **one** **room** or it is the **first** **room** increase the **capacity** with its current **id** number. If the room is **not** **reserved**, continue to the next one. You can increase the **capacity** of the **reserved room** if the previous one is **not reserved**.

**reserve\_first\_room()** **reserves** the **first** **room** in the database.

**delete\_last\_room()** **deletes** the **last** **room** in the database if it is **not** **reserved**.

### Examples

**This data is used as an example for the given test code below. You can populate the database using both methods of adding data with Django queries.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **room\_number** | **room\_type** | **capacity** | **amenities** | **price\_per\_night** |
| 401 | Standard | 2 | Tv | 100.00 |
| 501 | Deluxe | 3 | Wi-Fi | 200.00 |
| 601 | Deluxe | 6 | Jacuzzi | 400.00 |

**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** |
| print(get\_deluxe\_rooms()) reserve\_first\_room() print(HotelRoom.objects.get(room\_number=401).is\_reserved) |
| **Output** |
| Deluxe room with number 501 costs 200.00$ per night!  True |

## \*Character

Write a Django model called **"Character"** with the provided **fields**:

* **name** - character field, **consisting of a maximum of 100 characters**.
* **class\_name** - character field, **consisting** of a **maximum** of **20 characters**, with **choices - "Mage"**, **Warrior"**, **"Assassin"**, and **"Scout"**.
* **level** - positive integer field.
* **strength** - positive integer field.
* **dexterity** - positive integer field.
* **intelligence** - positive integer field.
* **hit\_points** - positive integer field.
* **inventory** - text field.

### Functions inside the caller.py file

**update\_characters()** **updates** every character based on their **class** **name**.

* If the class name is **"Mage"** - **increase** the **level** by 3 and **decrease** the **intelligence** by 7.
* If the class name is **"Warrior"** - **decrease** the hit **points** by half and **increase** the **dexterity** by 4.
* If the class name is **"Assassin"** or **"Scout"** - **update** their inventory to **"The inventory is empty"**.

**fuse\_characters(first\_character: Character, second\_character: Character)** **creates a** new **fusion** between 2 given characters. The new **mega-character** should be created with a specific combination between the **fields** of its **fusion** materials. The **fusion** characters must be **deleted** from the database and the **new** **character** must be **saved** in the database. The newly generated **fields** are:

* **name**
* **"{first\_character\_name} {second\_character\_name}"**.
* **class\_name**
* The class name should be set to "**Fusion**".
* **level**
* **(first\_character\_level + second\_character\_level) // 2**
* **strength**
* **(first\_character\_strength + second\_character\_strength) \* 1.2**
* **dexterity**
* **(first\_character\_dexterity + second\_character\_dexterity) \* 1.4**
* **intelligence**
* **(first\_character\_intelligence + second\_character\_intelligence) \* 1.5**

**Save the level, strength, dexterity, and intelligence as positive integers.**

* **hit\_points**
* **(first\_character\_hit\_points + second\_character\_hit\_points)**
* **inventory** - depending on the class of the **first** fusion character the inventory changes with different sets of items:
* For class name **"Mage"** or **"Scout"** - **"Bow of the Elven Lords, Amulet of Eternal Wisdom"**
* For class name **"Warrior"** or **"Assassin"** - **"Dragon Scale Armor, Excalibur"**

**grand\_dexterity()** **changes** the **dexterity** of every character to **30**.

**grand\_intelligence()** **changes** the **intelligence** of every character to **40**.

**grand\_strength()** **changes** the **strength** of every character to **50**.

**delete\_characters()** **deletes** all characters that have **inventory** with the text **"The inventory is empty"**.

### 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** |
| character1 = Character.objects.create(  name='Gandalf',  class\_name='Mage',  level=10,  strength=15,  dexterity=20,  intelligence=25,  hit\_points=100,  inventory='Staff of Magic, Spellbook',  )  character2 = Character.objects.create(  name='Hector',  class\_name='Warrior',  level=12,  strength=30,  dexterity=15,  intelligence=10,  hit\_points=150,  inventory='Sword of Troy, Shield of Protection',  )  fuse\_characters(character1, character2) fusion = Character.objects.filter(class\_name='Fusion').get()  print(fusion.name) print(fusion.class\_name) print(fusion.level) print(fusion.intelligence) print(fusion.inventory) |
| **Output** |
| Gandalf Hector  Fusion  11  52  Bow of the Elven Lords, Amulet of Eternal Wisdom |

### Hint

* You can use the **methods "filter()" and "update()"** when working with **queries**.