# Exercises: Working with Queries in Django

Submit your solutions to the SoftUni [**Judge system**](https://alpha.judge.softuni.org/Contests/Working-with-Queries-in-Django-Exercise/4326).

**Ask** **your** **questions** here: [https://www.slido.com](https://www.slido.com/) by entering the course code **#python-db**

For this exercise, you are given an **ORM project skeleton** (you can **download** it from the current lesson's resources) with **four models** - **ChessPlayer**, **Meal**, **Dungeon**, and **Workout**. The necessary **information** is described in every **exercise**.

## Artwork Gallery

Write a Django model called **"ArtworkGallery"** with the provided information:

* **artist\_name** - character field, **consisting of a maximum of 100 characters**.
* **art\_name** - character field, **consisting of a maximum of 100 characters**.
* **rating -** integer field.
* **price -** decimal field, with a **maximum** of 10 **digits** and 2 **decimal places**.

### Functions inside the caller.py file

**show\_highest\_rated\_art()** **returns** a string with the **highest-rated** art. If **two or more** **arts** have the same **rating**, the function should **return** the one with **the lowest id** from the database:

* **"{art\_name} is the highest-rated art with a {rating} rating!"**

**bulk\_create\_arts(first\_art: ArtworkGallery, second\_art: ArtworkGallery)** **bulk** **creates** two new **instances** of the **"ArtworkGallery"** class and **saves** them into the database.

**delete\_negative\_rated\_arts() deletes** all arts that have a **negative** **rating**. **0** (**zero**) counts as **positive**.

### 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** |
| artwork1 = ArtworkGallery(artist\_name='Vincent van Gogh', art\_name='Starry Night', rating=4, price=1200000.0)  artwork2 = ArtworkGallery(artist\_name='Leonardo da Vinci', art\_name='Mona Lisa', rating=5, price=1500000.0)  # Bulk saves the instances  bulk\_create\_arts(artwork1, artwork2)  print(show\_highest\_rated\_art())  print(ArtworkGallery.objects.all()) |
| **Output** |
| Mona Lisa is the highest-rated art with a 5 rating!  <QuerySet [<ArtworkGallery: ArtworkGallery object (1)>, <ArtworkGallery: ArtworkGallery object (2)>]> |

## Laptop

Write a Django model called **"Laptop"** with the provided information:

* **brand** - character field, consisting of a **maximum** of **20** **characters**, **with choices - "Asus", "Acer", "Apple", "Lenovo",** and **"Dell"**.
* **processor** - character field, **consisting of a maximum of 100 characters**.
* **memory** - positive integer field with **help text** **"Memory in GB"**.
* **storage** - positive integer field with **help text** **"Storage in GB"**.
* **operation\_system** - character field, **with choices - "Windows", "MacOS", "Linux", "Chrome OS".**
* **price** - decimal field with **maximum of 10 digits** and **2 decimal places.**

### Functions inside the caller.py file

**show\_the\_most\_expensive\_laptop()** **returns** a string with the **most expensive** laptop. If **two or more** **laptops** share the same **price**, the function should **return** the one with **the highest id** from the database.

* **"{brand} is the most expensive laptop available for {price}$!"**

**bulk\_create\_laptops(args: List[Laptop])** **bulk** **creates** one or more new **instances** of the **"Laptop"** class and **saves** them into the database. **The arguments will be given as instances in a list.**

**update\_to\_512\_GB\_storage()** **updates** the **storage** for allthe **"Asus"** and **"Lenovo" laptops** to **512** GB.

**update\_to\_16\_GB\_memory()** **updates** the **memory** for all **the "Apple", "Dell",** and **"Acer" laptops** to **16** GB.

**update\_operation\_systems()** **updates** the **operation system** for every laptop.

* If the **brand** is **"Asus"**, **update** the **operation** **system** to **"Windows"**.
* If the **brand** is **"Apple"**, **update** the **operation** **system** to **"MacOS"**.
* If the **brand** is **"Dell"** or **"Acer"**, **update** the **operation** **system** to **"Linux"**.
* If the **brand** is **"Lenovo"**, **update** the **operation** **system** to **"Chrome OS"**.

**delete\_inexpensive\_laptops()** **deletes** all **laptops** that have a **price** lessthan **1200**.

### 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** |
| laptop1 = Laptop(  brand='Asus',  processor='Intel Core i5',  memory=8,  storage=256,  operation\_system='MacOS',  price=899.99  )  laptop2 = Laptop(  brand='Apple',  processor='Chrome OS',  memory=16,  storage=256,  operation\_system='MacOS',  price=1399.99  )  laptop3 = Laptop(  brand='Lenovo',  processor='AMD Ryzen 7',  memory=12,  storage=256,  operation\_system='Linux',  price=999.99,  )  # Create a list of instances  laptops\_to\_create = [laptop1, laptop2, laptop3]  # Use bulk\_create to save the instances  bulk\_create\_laptops(laptops\_to\_create)  update\_to\_512\_GB\_storage()  update\_operation\_systems()  # Retrieve 2 laptops from the database  asus\_laptop = Laptop.objects.filter(brand\_\_exact='Asus').get()  lenovo\_laptop = Laptop.objects.filter(brand\_\_exact='Lenovo').get()  print(asus\_laptop.storage)  print(lenovo\_laptop.operation\_system) |
| **Output** |
| 512  Chrome OS |

## Chess Player

For this exercise, you are going to use the **already-configured** Django model called **"ChessPlayer"**. The model has the following **fields**: **username**, **title**, **rating**, **games\_played**, **games\_won**, **games\_lost**, and **games\_drawn**.

**Note: Do not forget to apply the necessary changes (migrations) to the database.**

### Functions inside the caller.py file

**bulk\_create\_chess\_players(args: List[ChessPlayer])** **bulk creates** one or more new **instances** of the model **"ChessPlayer"** class and **saves** them into the database. **The arguments will be given as instances in a list.**

**delete\_chess\_players() deletes** all the chess **players** that have "**no** **title"**.

**change\_chess\_games\_won() changes** the games **won** for the players with a "**GM"** **title** to **30**.

**change\_chess\_games\_lost()** **changes** the games **lost** for the players with "**no title"** to **25**.

**change\_chess\_games\_drawn() changes** the games **drawn** for **every** player to **10**.

**grand\_chess\_title\_GM()** **changes** the **title** to **"GM"** for every player with a **rating** greater than or equal to **2400**.

**grand\_chess\_title\_IM()** **changes** the **title** to **"IM"** for every player with a **rating** between **2399** and **2300** (**both inclusive**).

**grand\_chess\_title\_FM()** **changes** the **title** to **"FM"** for every player with a **rating** between **2299** and **2200** (**both inclusive**).

**grand\_chess\_title\_regular\_player()** **changes** the **title** to **"regular player"** for every player with a **rating** between **2199** and **0** (**both inclusive**).

### 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** |
| player1 = ChessPlayer(  username='Player1',  title='no title',  rating=2200,  games\_played=50,  games\_won=20,  games\_lost=25,  games\_drawn=5, ) player2 = ChessPlayer(  username='Player2',  title='IM',  rating=2350,  games\_played=80,  games\_won=40,  games\_lost=25,  games\_drawn=15, )  *# Call the bulk\_create\_chess\_players function* bulk\_create\_chess\_players([player1, player2])  *# Call the delete\_chess\_players function* delete\_chess\_players()  *# Check that the players are deleted* print("Number of Chess Players after deletion:", ChessPlayer.objects.count()) |
| **Output** |
| Number of Chess Players after deletion: 1 |

## Meal [Solve with AI]

For this exercise, you are going to use the **already-configured** Django model called **"Meal"**. The model has the following fields: **name**, **meal\_type**, **preparation\_time**, **difficulty**, **calories**, and **chef**.

**Note: Do not forget to apply all the migrations to the database.**

### Functions inside the caller.py file

**set\_new\_chefs()** **updates** the **name** for every **meal**.

* If the meal **type** is **"Breakfast"**, **update** the **chef's name** to **"Gordon Ramsay"**.
* If the meal **type** is **"Lunch"**, **update** the **chef's name** to **"Julia Child"**.
* If the meal **type** is **"Dinner"**, **update** the **chef's name** to **"Jamie Oliver"**.
* If the meal **type** is **"Snack"**, **update** the **chef's name** to **"Thomas Keller"**.

**set\_new\_preparation\_times()** **updates** the **preparation time** for every **meal**.

* If the meal **type** is **"Breakfast"**, **update** the **preparation** **time** to "**10 minutes"**.
* If the meal **type** is **"Lunch"**, **update** the **preparation** **time** to "**12 minutes"**.
* If the meal **type** is **"Dinner"**, **update** the **preparation** **time** to "**15 minutes".**
* If the meal **type** is **"Snack"**, **update** the **preparation** **time** to "**5 minutes"**.

**update\_low\_calorie\_meals()** **changes** the **calories** for the **"Breakfast"** and the **"Dinner"** **meals** to **400**.

**update\_high\_calorie\_meals()** **changes** the **calories** for the **"Lunch"** and the **"Snack" meals** to **700**.

**delete\_lunch\_and\_snack\_meals()** **deletes** all the **meals** from the **types** **"Lunch"** and **"Snack"**.

### 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** |
| meal1 = Meal.objects.create(  name="Pancakes",  meal\_type="Breakfast",  preparation\_time="20 minutes",  difficulty=3,  calories=350,  chef="Jane", )  meal2 = Meal.objects.create(  name="Spaghetti Bolognese",  meal\_type="Dinner",  preparation\_time="45 minutes",  difficulty=4,  calories=550,  chef="Sarah", )  *# Test the set\_new\_chefs function* set\_new\_chefs()  *# Test the set\_new\_preparation\_times function* set\_new\_preparation\_times()  *# Refreshes the instances* meal1.refresh\_from\_db() meal2.refresh\_from\_db()  *# Print the updated meal information* print("Meal 1 Chef:", meal1.chef) print("Meal 1 Preparation Time:", meal1.preparation\_time) print("Meal 2 Chef:", meal2.chef) print("Meal 2 Preparation Time:", meal2.preparation\_time) |
| **Output** |
| Meal 1 Chef: Gordon Ramsay  Meal 1 Preparation Time: 10 minutes  Meal 2 Chef: Jamie Oliver  Meal 2 Preparation Time: 15 minutes |

## Dungeon [Solve with AI]

For this exercise, you are going to use the **already-configured** Django model called **"Dungeon"**. The model has the following fields: **name**, **difficulty**, **location**, **boss\_name**, **recommended\_level**, **boss\_health**, and **reward**.

**Note: Do not forget to apply all the migrations to the database.**

### Functions inside the caller.py file

**show\_hard\_dungeons()** **returns** a string with only the **"Hard"** dungeons, **ordered by** **location** (**descending**):

* **"{dungeon\_name\_1} is guarded by {boss\_1} who has {health\_1} health points!**

**…**

**{dungeon\_name\_N} is guarded by {boss\_N} who has {health\_N} health points!"**

**bulk\_create\_dungeons(args: List[Dungeon])** **creates** one or more new **instances** of the **"Dungeon"** class and **saves** them into the database. **The arguments will be given as instances in a list.**

**update\_dungeon\_names()** **updates** the **name** for all dungeons.

* If the dungeon **difficulty** is **"Easy"**, **update** the dungeon **name** to **"The Erased Thombs"**.
* If the dungeon **difficulty** is **"Medium"**, **update** the dungeon **name** to **"The Coral Labyrinth"**.
* If the dungeon **difficulty** is **"Hard"**, **update** the dungeon **name** to **"The Lost Haunt"**.

**update\_dungeon\_bosses\_health()** **changes** the **boss** **health** to **500** for all dungeons **except** for the ones that have **difficulty** **"Easy"**.

**update\_dungeon\_recommended\_levels()** **updates** the **recommended level** for all dungeons.

* If the dungeon **difficulty** is **"Easy"**, **update** the **recommended** level to **25**.
* If the dungeon **difficulty** is **"Medium"**, **update** the **recommended** level to **50**.
* If the dungeon **difficulty** is **"Hard"**, **update** the **recommended** level to **75**.

**update\_dungeon\_rewards()** **updates** the **difficulty** for all dungeons.

* If the dungeon **boss's health** is **500**, **update** the dungeon **reward** to **"1000 Gold".**
* If the dungeon's **location** starts with "**E**", **update** the **reward** to **"New dungeon unlocked"**.
* If the dungeon's **location** ends with "**s**", **update** the **reward** to **"Dragonheart Amulet"**.

**set\_new\_locations()** **updates** the **location** for all dungeons.

* If the **recommended** level is **25**, **update** the dungeon **location** to **"Enchanted Maze"**.
* If the **recommended** level is **50**, **update** the dungeon **location** to **"Grimstone Mines".**
* If the **recommended** level is **75**, **update** the dungeon **location** to **"Shadowed Abyss"**.

### 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 two instances dungeon1 = Dungeon(  name="Dungeon 1",  boss\_name="Boss 1",  boss\_health=1000,  recommended\_level=75,  reward="Gold",  location="Eternal Hell",  difficulty="Hard", )  dungeon2 = Dungeon(  name="Dungeon 2",  boss\_name="Boss 2",  boss\_health=400,  recommended\_level=25,  reward="Experience",  location="Crystal Caverns",  difficulty="Easy", )  # Bulk save the instances bulk\_create\_dungeons([dungeon1, dungeon2])  # Update boss's health update\_dungeon\_bosses\_health()  # Show hard dungeons hard\_dungeons\_info = show\_hard\_dungeons() print(hard\_dungeons\_info)  # Change dungeon names based on difficulty update\_dungeon\_names() dungeons = Dungeon.objects.order\_by('boss\_health') print(dungeons[0].name) print(dungeons[1].name)  # Change the dungeon rewards update\_dungeon\_rewards() dungeons = Dungeon.objects.order\_by('boss\_health') print(dungeons[0].reward) print(dungeons[1].reward) |
| **Output** |
| Dungeon 1 is guarded by Boss 1 who has 500 health points!  The Erased Thombs  The Lost Haunt  Dragonheart Amulet  New dungeon unlocked |

## Workout [Solve with AI]

For this exercise, you are going to use the **already-configured** Django model called **"Workout"**. The model has the following fields: **name**, **workout\_type**, **duration**, **difficulty**, **calories\_burned**, and **instructor**.

### Functions inside the caller.py file

**show\_workouts()** **returns** only the **"Calisthenics"** and **"CrossFit"** workouts with their **name**, **type,** and **difficulty, ordered by id** (**ascending**) as a string as follows:

* **"{workout\_name\_1} from {workout\_type\_1} type has {difficulty\_1} difficulty!**

**…**

**{workout\_name\_N} from {workout\_type\_N} type has {difficulty\_N} difficulty!"**

**get\_high\_difficulty\_cardio\_workouts()** **returns** all workouts **from type** **"Cardio"** that have **difficulty** **"High"**, **ordered by** the **instructor**.

**set\_new\_instructors()** **updates** the **instructors** for all workouts.

* If the **workout** **type** is **"Cardio"**, **update** the **instructor** to **"John Smith"**.
* If the **workout** **type** is **"Strength"**, **update** the **instructor** to **"Michael Williams"**.
* If the **workout** **type** is **"Yoga"**, **update** the **instructor** to **"Emily Johnson"**.
* If the **workout** **type** is **"CrossFit"**, **update** the **instructor** to **"Sarah Davis"**.
* If the **workout** **type** is **"Calisthenics"**, **update** the **instructor** to **"Chris Heria"**.

**set\_new\_duration\_times()** **updates** the **instructor** for every workout.

* If the **instructor** is **"John Smith"**, **update** the **duration** **time** to "**15 minutes"**.
* If the **instructor** is **"Sarah Davis"**, **update** the **duration** **time** to "**30 minutes"**.
* If the **instructor** is **"Chris Heria"**, **update** the **duration** **time** to "**45 minutes"**.
* If the **instructor** is **"Michael Williams"**, **update** the **duration** **time** to "**1 hour"**.
* If the **instructor** is **"Emily Johnson"**, **update** the **duration** **time** to "**1 hour and** **30 minutes"**.

**delete\_workouts()** **deletes** all workouts **except** the **"Strength"** and **"Calisthenics"**.

### 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 two Workout instances  workout1 = Workout.objects.create(  name="Push-Ups",  workout\_type="Calisthenics",  duration="10 minutes",  difficulty="Intermediate",  calories\_burned=200,  instructor="Bob"  )  workout2 = Workout.objects.create(  name="Running",  workout\_type="Cardio",  duration="30 minutes",  difficulty="High",  calories\_burned=400,  instructor="Lilly"  )  # Run the functions  print(show\_workouts())  high\_difficulty\_cardio\_workouts = get\_high\_difficulty\_cardio\_workouts()  for workout in high\_difficulty\_cardio\_workouts:  print(f"{workout.name} by {workout.instructor}")  set\_new\_instructors()  for workout in Workout.objects.all():  print(f"Instructor: {workout.instructor}")  set\_new\_duration\_times()  for workout in Workout.objects.all():  print(f"Duration: {workout.duration}") |
| **Output** |
| Push-Ups from Calisthenics type has Intermediate difficulty!  Running by Lilly  Instructor: Chris Heria  Instructor: John Smith  Duration: 45 minutes  Duration: 15 minutes |