# Exercises: Advanced Queries in Django

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/4334/Advanced-Queries-in-Django-Exercise).

For this exercise, you are given an **ORM project skeleton** (you can download it from the current lesson's resources) with **9 models.** The necessary **information** is described in every **exercise**.

## Real Estate Listing

Create a custom model manager for a **real estate application** that displays various **real estate properties**. Your objective is to build **advanced filtering functionalities** to enhance **user** **property** **search** **experiences**.

### Model RealEstateListing

Use the **already-configured** Django model called "**RealEstateListing"** with the provided **fields**:

* **property\_type**, **price**, **bedrooms**, and **location**.
* **objects** - custom manager - **"RealEstateListingManager()".**

### Manager RealEstateListingManager

Create a **custom manager** that handles different types of **queries**. The custom manager has the following **methods**:

**by\_property\_type(property\_type: str)** - **returns** all real estate **objects (in a queryset)** from the given **property type**.

**in\_price\_range(min\_price: Decimal, max\_price: Decimal)** - **returns** all real estate **objects (in a queryset)** between the **given price range** (**inclusive**).

**with\_bedrooms(bedrooms\_count: int)** - **returns** all real estate **objects (in a queryset)** with the given **bedroom count**.

**popular\_locations()** - **returns** the **2** most visited **locations, ordered by location alphabetically (ascending)**. The most visited locations are those with **the most database records**.

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create instances of RealEstateListing with locations RealEstateListing.objects.create(  property\_type='House',  price=100000.00,  bedrooms=3,  location='Los Angeles' )  RealEstateListing.objects.create(  property\_type='Flat',  price=75000.00,  bedrooms=2,  location='New York City' )  RealEstateListing.objects.create(  property\_type='Villa',  price=250000.00,  bedrooms=4,  location='Los Angeles' # Same location as the first instance )  RealEstateListing.objects.create(  property\_type='House',  price=120000.00,  bedrooms=3,  location='San Francisco' )  # Run the 'by\_property\_type' method house\_listings = RealEstateListing.objects.by\_property\_type('House') print("House listings:") for listing in house\_listings:  print(f"- {listing.property\_type} in {listing.location}")  # Run the 'in\_price\_range' method affordable\_listings = RealEstateListing.objects.in\_price\_range(75000.00, 120000.00) print("Price in range listings:") for listing in affordable\_listings:  print(f"- {listing.property\_type} in {listing.location}")  # Run the 'with\_bedrooms' method two\_bedroom\_listings = RealEstateListing.objects.with\_bedrooms(2) print("Two-bedroom listings:") for listing in two\_bedroom\_listings:  print(f"- {listing.property\_type} in {listing.location}")  # Run the 'popular\_locations' method popular\_locations = RealEstateListing.objects.popular\_locations() print("Popular locations:") for location in popular\_locations:  print(f"- {location['location']} ; Listings: {location['location\_count']}") |
| **Output** |
| House listings:  - House in Los Angeles  - House in San Francisco  Price in range listings:  - House in Los Angeles  - Flat in New York City  - House in San Francisco  Two-bedroom listings:  - Flat in New York City  Popular locations:  - Los Angeles; Listings: 2  - New York City; Listings: 1 |

## Video Games Library

### You'll create a range of tools to explore the world of video games, allowing users to discover top-rated titles, filter games by release dates, and gain a deeper understanding of popular genres. These methods will empower users to make informed decisions and find the perfect gaming experiences tailored to their preferences.

### Model VideoGame

Use the **already-configured** Django model called "**VideoGame"** with the provided **fields**:

* **title**, **genre**, **rating**, and **release\_year**.
* **objects**" - custom manager - **"VideoGameManager()"**.

For some of them, you should apply several **validations**.

* **rating** - the **rating** must be between **0.0** and **10.0 (both inclusive)**, otherwise **raise** a **ValidationError** with the message: **"The rating must be between 0.0 and 10.0"**.
* **release\_year** - the release year must be between **1990** and **2023 (both inclusive)**, otherwise **raise** a **ValidationError** with the message: **"The release year must be between 1990 and 2023".**

### Manager VideoGameManager

Create a **custom** **manager** that handles different types of **queries**. The custom manager has the following **methods**:

**games\_by\_genre(genre: str)** - **returns** all game **objects (in a queryset)** from the given **genre**.

**recently\_released\_games(year: int)** - **returns** all game **objects (in a queryset)** that are **released** **after or in the same year** as the given **year**.

**highest\_rated\_game()** - **returns** the **highest-rated** game.

**lowest\_rated\_game()** - **returns** the **lowest-rated** game.

**average\_rating()** - **returns** the **calculation** of the **average rating** of all **games** in the database, **formatted** to the **first** decimal place, **ordered by** the **average rating** (**descending**).

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create instances of VideoGame with real data game1 = VideoGame.objects.create(title="The Last of Us Part II", genre="Action", release\_year=2020, rating=9.0)  game2 = VideoGame.objects.create(title="Cyberpunk 2077", genre="RPG", release\_year=2020, rating=7.2)  game3 = VideoGame.objects.create(title="Red Dead Redemption 2", genre="Adventure", release\_year=2018, rating=9.7)  game4 = VideoGame.objects.create(title="FIFA 22", genre="Sports", release\_year=2021, rating=8.5)  game5 = VideoGame.objects.create(title="Civilization VI", genre="Strategy", release\_year=2016, rating=8.8)  # Run the custom manager methods action\_games = VideoGame.objects.games\_by\_genre('Action') recent\_games = VideoGame.objects.recently\_released\_games(2019) average\_rating = VideoGame.objects.average\_rating() highest\_rated = VideoGame.objects.highest\_rated\_game() lowest\_rated = VideoGame.objects.lowest\_rated\_game()  # Print the results print(action\_games) print(recent\_games) print(average\_rating) print(highest\_rated) print(lowest\_rated) |
| **Output** |
| <QuerySet [<VideoGame: The Last of Us Part II>]>  <QuerySet [<VideoGame: The Last of Us Part II>, <VideoGame: Cyberpunk 2077>, <VideoGame: FIFA 22>, <VideoGame: The Last of Us Part II>, <VideoGame: Cyberpunk 2077>, <VideoGame: FIFA 22>]>  8.6  Red Dead Redemption 2  Cyberpunk 2077 |

## Shopaholic Haven

Imagine you're the lead developer for a dynamic e-commerce platform named "**Shopaholic Haven**". This platform manages many product orders, each intricately tied to customer profiles. To ensure that your platform delivers a lightning-fast shopping experience, optimizing how you retrieve and display order information along with the corresponding customer details is imperative.

### Model BillingInfo

Use the **already-configured** Django model called "**BillingInfo"** with the provided **field**:

* **address**.

### Model Invoice

Use the **already-configured** Django model called "**Invoice"** with the provided **fields**:

* **invoice\_number** and **billing\_info**.

### Methods inside the Invoice model

Implement the following methods and execute them with specific **methods** that **reduce** the **number** of database **queries** and improve **performance**:

**get\_invoices\_with\_prefix(prefix: str)** - **returns** all the invoices (**in a queryset**), starting with the specific **prefix** in the **invoice number**.

**get\_invoices\_sorted\_by\_number()** - **returns** all the invoices (**in a queryset**), **ordered** by **invoice number** (**ascending**)

**get\_invoice\_with\_billing\_info(invoice\_number: str)** - **returns** the invoice **object** by a specific **invoice number**.

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create BillingInfo instances with real addresses billing\_info\_1 = BillingInfo.objects.create(address="456 Oak Lane, Boston, MA 02108")  billing\_info\_2 = BillingInfo.objects.create(address="789 Maple Avenue, San Francisco, CA 94101")  billing\_info\_3 = BillingInfo.objects.create(address="101 Pine Street, New York, NY 10001")  # Create Invoice instances with related BillingInfo invoice\_1 = Invoice.objects.create(invoice\_number="INV007", billing\_info=billing\_info\_1)  invoice\_2 = Invoice.objects.create(invoice\_number="INV002", billing\_info=billing\_info\_2) invoice\_3 = Invoice.objects.create(invoice\_number="INV004", billing\_info=billing\_info\_3)  # Get invoices starting with a specific prefix  invoices\_with\_prefix = Invoice.get\_invoices\_with\_prefix("INV")  for invoice in invoices\_with\_prefix:  print(f"Invoice Number with prefix INV: {invoice.invoice\_number}")  # Get invoices sorted by invoice number invoices\_sorted = Invoice.get\_invoices\_sorted\_by\_number()  for invoice in invoices\_sorted:  print(f"Invoice Number: {invoice.invoice\_number}")  # Get an invoice by a specific invoice number along with its related billing info invoice = Invoice.get\_invoice\_with\_billing\_info("INV002") print(f"Invoice Number: {invoice.invoice\_number}") print(f"Billing Info: {invoice.billing\_info.address}") |
| **Output** |
| Invoice Number with prefix INV: INV007  Invoice Number with prefix INV: INV002  Invoice Number with prefix INV: INV004  Invoice Number: INV002  Invoice Number: INV004  Invoice Number: INV007  Invoice Number: INV002  Billing Info: 789 Maple Avenue, San Francisco, CA 94101 |

## 4. IT Sector

In this exercise, you will work with a Django project featuring three models: "**Technology"**, "**Project"**, and "**Programmer"**. The objective is to optimize database queries, reduce database hits, and enhance performance.

### Model Technology

Use the **already-configured** Django model called "**Technology"** with the provided **fields**:

* **name** and **description**.

### Model Project

Use the **already-configured** Django model called "**Project"** with the provided **fields**:

* **name**, **description**, and **technologies\_used**.

### Model Programmer

Use the **already-configured** Django model called "**Programmer"** with the provided **fields**:

* **name** and **projects.**

### Methods inside the Project model

Implement the following methods and execute them with specific **methods** that **reduce** the **number** of database **queries** and improve **performance**:

**get\_programmers\_with\_technologies()** - **returns** all **programmers** and all **technologies**, related to the **project** (**in a queryset**).

### Methods inside the Programmer model

Implement the following methods and execute them with specific **methods** that **reduce** the **number** of database **queries** and improve the **performance**:

**get\_projects\_with\_technologies()** - **returns** all **projects** and all **technologies (for the current project)**, related to the **programmer** (**in a queryset**).

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create instances of Technology tech1 = Technology.objects.create(name="Python", description="A high-level programming language")  tech2 = Technology.objects.create(name="JavaScript", description="A scripting language for the web")  tech3 = Technology.objects.create(name="SQL", description="Structured Query Language")  # Create instances of Project project1 = Project.objects.create(name="Web App Project", description="Developing a web application")  project1.technologies\_used.add(tech1, tech2)  project2 = Project.objects.create(name="Database Project", description="Managing databases")  project2.technologies\_used.add(tech3)  # Create instances of Programmer programmer1 = Programmer.objects.create(name="Alice") programmer2 = Programmer.objects.create(name="Bob")  # Associate projects with programmers programmer1.projects.add(project1, project2) programmer2.projects.add(project1)  # Execute the "get\_programmers\_with\_technologies" method for a specific project specific\_project = Project.objects.get(name="Web App Project") programmers\_with\_technologies = specific\_project.get\_programmers\_with\_technologies()  # Iterate through the related programmers and technologies for programmer in programmers\_with\_technologies:  print(f"Programmer: {programmer.name}")  for technology in programmer.projects.get(name="Web App Project").technologies\_used.all():  print(f"- Technology: {technology.name}")  # Execute the "get\_projects\_with\_technologies" method for a specific programmer specific\_programmer = Programmer.objects.get(name="Alice") projects\_with\_technologies = specific\_programmer.get\_projects\_with\_technologies()  # Iterate through the related projects and technologies for project in projects\_with\_technologies:  print(f"Project: {project.name} for {specific\_programmer.name}")  for technology in project.technologies\_used.all():  print(f"- Technology: {technology.name}") |
| **Output** |
| Programmer: Alice  - Technology: Python  - Technology: JavaScript  Programmer: Bob  - Technology: Python  - Technology: JavaScript  Project: Web App Project for Alice  - Technology: Python  - Technology: JavaScript  Project: Database Project for Alice  - Technology: SQL |

## 5. Taskify

In this exercise, you'll develop a Django model for task management with complex queries. The "**Task"** model includes various fields such as title, description, priority, assignment, completion status, and timestamps. The goal is to implement a set of class methods that allow for advanced task management and querying.

### Model Task

Use the **already-configured** Django model called "**Task"** with the provided **fields**:

* **title**, **description**, **priority**, **is\_completed**, **creation\_date**, **completion\_date**.

### Methods inside the Task model

Implement the following methods and execute them with specific **methods** that **reduce** the **number** of database **queries** and improve **performance**:

**ongoing\_high\_priority\_tasks()** - **returns** all **tasks (in a query set) that:**

* Have priority set to "**High**".
* Are **not** **completed**.
* Have a completion date **greater than** the creation date.

**completed\_mid\_priority\_tasks()** - **returns** all **tasks (in a queryset) that:**

* Have priority set to "**Medium**".
* Are **completed**.

**search\_tasks(query: str)** - **returns** all **tasks (in a queryset) that:**

* Contain the **query** in their **title** or their **description**.

**recent\_completed\_tasks(days: int)** - **returns** all **tasks (in a queryset) that:**

* Are **completed**.
* Have a completion date **greater than or equal** to the creation date **subtracted** by the given **days**.

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create task instances with custom creation dates task1 = Task(  title="Task 1",  description="Description for Task 1",  priority="High",  creation\_date=date(2023, 1, 15),  completion\_date=date(2023, 1, 25) )  task2 = Task(  title="Task 2",  description="Description for Task 2",  priority="Medium",  is\_completed=True,  creation\_date=date(2023, 2, 1),  completion\_date=date(2023, 2, 10) )  task3 = Task(  title="Task 3",  description="Description for Task 3",  priority="Hard",  is\_completed=True,  creation\_date=date(2023, 1, 15),  completion\_date=date(2023, 1, 20) )  # Save the tasks to the database task1.save() task2.save() task3.save()  # Now, you can run the defined methods  # 1. Get ongoing high-priority tasks ongoing\_high\_priority = Task.ongoing\_high\_priority\_tasks() print("Ongoing High Priority Tasks:") for task in ongoing\_high\_priority:  print('- ' + task.title)  # 2. Get completed medium-priority tasks completed\_mid\_priority = Task.completed\_mid\_priority\_tasks() print("Completed Medium Priority Tasks:") for task in completed\_mid\_priority:  print('- ' + task.title)  # 3. Search for tasks based on a query search\_results = Task.search\_tasks("Task 3") print("Search Results:") for task in search\_results:  print('- ' + task.title)  # 4. Get recent completed tasks recent\_completed = task1.recent\_completed\_tasks(days=5) print("Recent Completed Tasks:") for task in recent\_completed:  print('- ' + task.title) |
| **Output** |
| Ongoing High Priority Tasks:  - Task 1  Completed Medium Priority Tasks:  - Task 2  Search Results:  - Task 3  Recent Completed Tasks:  - Task 2  - Task 3 |

## 6. Gym Session

### Model Exercise

Use the **already-configured** Django model called "**Exercise"** with the provided **fields**:

* "**name**", "**category**", "**difficulty\_level**", "**duration\_minutes** ", and "**repetitions**".

### Methods inside the Exercise model

Implement the following methods and execute them with specific **methods** that **reduce** the **number** of database **queries** and improve **performance**:

**get\_long\_and\_hard\_exercises()** - **returns** all **exercises (in a queryset) that:**

* Duration minutes **greater than** **30**.
* Difficulty **greater than or equal to 10**.

**get\_short\_and\_easy\_exercises()** - **returns** all **exercises (in a queryset) that:**

* Duration minutes **less than** **15**.
* Difficulty **less than 5**.

**get\_exercises\_within\_duration(min\_duration: int, max\_duration: int)** - **returns** all **exercises (in a queryset) that:**

* Duration minutes **greater than** **or equal to** **the** minimum duration.
* Duration minutes **less than or equal to** **the** maximum duration.

**get\_exercises\_with\_difficulty\_and\_repetitions(min\_difficulty: int, min\_repetitions: int)** - **returns** all **exercises (in a queryset) that:**

* Difficulty **greater than or equal to the** minimum difficulty.
* Repetitions **greater than or equal to the** minimum repetitions.

### Examples

|  |
| --- |
| **Test Code - caller.py** |
| # Create instances of Exercise exercise1 = Exercise.objects.create(  name="Push-ups",  category="Strength",  difficulty\_level=4,  duration\_minutes=10,  repetitions=50, )  exercise2 = Exercise.objects.create(  name="Running",  category="Cardio",  difficulty\_level=7,  duration\_minutes=20,  repetitions=0, )  exercise3 = Exercise.objects.create(  name="Pull-ups",  category="Strength",  difficulty\_level=13,  duration\_minutes=35,  repetitions=20, )  # Print the results long\_and\_hard\_exercises = Exercise.get\_long\_and\_hard\_exercises() print("Long and hard exercises:") for exercise in long\_and\_hard\_exercises:  print('- ' + exercise.name)  short\_and\_easy\_exercises = Exercise.get\_short\_and\_easy\_exercises() print("Short and easy exercises:") for exercise in short\_and\_easy\_exercises:  print('- ' + exercise.name)  exercises\_within\_duration = Exercise.get\_exercises\_within\_duration(20, 40) print(f"Exercises within 20 - 40 minutes:") for exercise in exercises\_within\_duration:  print('- ' + exercise.name)  exercises\_with\_difficulty\_and\_repetitions = Exercise.get\_exercises\_with\_difficulty\_and\_repetitions(6, 15) print(f"Exercises with difficulty 6+ and repetitions 15+:") for exercise in exercises\_with\_difficulty\_and\_repetitions:  print('- ' + exercise.name) |
| **Output** |
| Long and hard exercises:  - Pull-ups  Short and easy exercises:  - Push-ups  Exercises within 20 - 40 minutes:  - Running  - Pull-ups  Exercises with difficulty 6+ and repetitions 15+:  - Pull-ups |