Question 1: By default are django signals executed synchronously or asynchronously? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

ANS - By default, Django signals are executed **synchronously**. This means that the signal handlers (the functions connected to signals) are executed during the normal flow of the request, and the request will not continue until the signal handler has finished executing.

To demonstrate this, we can create a Django signal handler and introduce an artificial delay (using `time.sleep`) within the handler to simulate a long-running task. We'll see that the delay directly impacts the request's response time, confirming that the signals are synchronous by default.

CODE -

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
import time

class MyModel(models.Model):
    name = models.CharField(max_length=255)

# Signal handler
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print(f"Signal received for: {instance.name}")
    time.sleep(5)  # Simulate a long-running task
    print("Signal handler completed")

# Now let's simulate saving an instance in the Django shell or a view:

# views.py
from django.http import HttpResponse
from .models import MyModel
```

```
def save_model(request):
    instance = MyModel.objects.create(name="Test Model")
    return HttpResponse("Model saved!")
# After setting up the views, run the server and access the URL associated with `save_model`.
# Expected output (server log):
# Signal received for: Test Model
# *waits for 5 seconds*
# Signal handler completed
# Response: "Model saved!" will be sent only after the 5-second delay
```

Explanation:

- When `MyModel.objects.create(name="Test Model")` is called in the view, the `post_save` signal is triggered.
- The signal handler `my_signal_handler` executes immediately, and the 5-second sleep causes the entire request to wait before the handler completes.
- The response from the view (`"Model saved!"`) is delayed until the signal handler finishes executing.

This behavior proves that the signals are executed synchronously by default in Django. If Django signals were asynchronous by default, the sleep in the signal handler would not block the request/response cycle.

Question 2: Do django signals run in the same thread as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

ANS - Yes, by default, Django signals run in the **same thread** as the caller. This means that the signal handler is executed in the same thread as the code that triggered the signal (for example, when a model's `save` method is called, and the corresponding signal handler is invoked).

To prove that signals run in the same thread, we can compare the thread IDs of the caller and the signal handler. Both should be the same if they are executed in the same thread.

Code Snippet:

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
import threading

class MyModel(models.Model):
    name = models.CharField(max_length=255)
```

```
# Signal handler
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    # Print the thread ID in the signal handler
    print(f"Signal handler thread ID: {threading.get_ident()}")
    print(f"Signal received for: {instance.name}")

# views.py
from django.http import HttpResponse
from .models import MyModel
import threading

def save_model(request):
    # Print the thread ID in the view (the caller)
    print(f"View thread ID: {threading.get_ident()}")
    instance = MyModel.objects.create(name="Test Model")
    return HttpResponse("Model saved!")

# Expected output in the console:
# View thread ID: 123456789 (example)
# Signal handler thread ID: 123456789 (same as the view thread ID)
# Signal received for: Test Model
```

Explanation:

- In the view (`save_model`), we create an instance of `MyModel`, which triggers the `post_save` signal.
- Both the view and the signal handler print the thread ID using `threading.get_ident()`.
- If Django signals run in the same thread as the caller, the thread ID printed in both the view and the signal handler should be identical.

Expected Output in the Console:

```
View thread ID: 139837517780032
Signal handler thread ID: 139837517780032
Signal received for: Test Model
```

Both the view and the signal handler have the same thread ID, conclusively proving that Django signals, by default, run in the same thread as the caller.

Question 3: By default do django signals run in the same database transaction as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

ANS - By default, Django signals run in the **same database transaction** as the caller. This means that if the caller is within a database transaction and the signal handler makes changes to the database, those changes will either be committed or rolled back along with the caller's transaction.

To demonstrate this, we can create a scenario where a database operation happens within a signal handler, and then we raise an exception in the caller after the signal is executed but before the transaction is committed. If the signal runs in the same transaction, the changes made by the signal handler should be rolled back when the caller's transaction is rolled back.

Code Snippet:

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

class MyModel(models.Model):
    name = models.CharField(max_length=255)

class LogEntry(models.Model):
    message = models.CharField(max_length=255)

# Signal handler
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    # Log a message in the database when a MyModel instance is saved
    LogEntry.objects.create(message=f"Created MyModel instance with name: {instance.name}")
    print("Signal handler executed, log entry created")

# views.py
from django.http import HttpResponse
from django.db import transaction
from .models import MyModel
```

```
def save_model_with_error(request):
    try:
        with transaction.atomic():
            # Create an instance of MyModel (which will trigger the signal)
            instance = MyModel.objects.create(name="Test Model")
            # Force an error after signal execution but before transaction commit
            raise Exception("Forcing rollback after signal execution")
    except Exception as e:
        return HttpResponse(f"Error occurred: {str(e)}")

    return HttpResponse("Model saved!")
```

Test:

1. Create a model instance of `MyModel` in a view, which triggers the `post_save` signal.
2. The signal handler creates a `LogEntry` instance in the database, logging the creation of the `MyModel` instance.
3. After the signal handler executes, we raise an exception in the view to force a rollback of the transaction.
4. Check whether the `LogEntry` created by the signal handler is still in the database after the exception is raised.

Explanation:
- If the signal runs in the same transaction as the caller, the `LogEntry` created by the signal handler will be rolled back when the transaction is rolled back due to the raised exception.
- If the signal runs in a different transaction, the `LogEntry` would persist in the database even after the exception is raised.

Expected Behavior:

1. The `LogEntry` created by the signal handler should **not** be present in the database after the exception is raised, because the entire transaction (including the signal handler's changes) should be rolled back.

Verifying with Database Query (Django Shell):

After calling the `save_model_with_error` view:
```python
from myapp.models import LogEntry
```

```
# Check if the log entry is present
LogEntry.objects.all()  # Should return an empty queryset if signal ran in the same transaction
```

Expected Output in the Database:

```python
<QuerySet []>  No entries in the LogEntry table, proving the transaction was rolled back.
```

Conclusion:

This proves that Django signals run in the same database transaction as the caller by default. If the transaction is rolled back, any changes made by the signal handler are also rolled back.

Topic: Custom Classes in Python

Description: You are tasked with creating a Rectangle class with the following requirements:

1. An instance of the `Rectangle` class requires `length:int` and `width:int` to be initialized.
2. We can iterate over an instance of the `Rectangle` class
3. When an instance of the `Rectangle` class is iterated over, we first get its length in the format: **`{'length': <VALUE_OF_LENGTH>}`** followed by the width **`{width: <VALUE_OF_WIDTH>}`**

ANS -

To meet the requirements, we need to create a custom `Rectangle` class that accepts `length` and `width` during initialization and allows iteration over the instance. When iterating, the `Rectangle` should first return a dictionary containing the length and then a dictionary containing the width.

We can achieve this by implementing the `__iter__` and `__next__` methods, allowing the class to behave like an iterable.

Implementation:

```
class Rectangle:
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width
        self._attributes = [{'length': self.length}, {'width':
self.width}]
        self._index = 0  # This will keep track of the current
position for iteration

    def __iter__(self):
        # This method is called when iteration is initiated (e.g., in
a for loop).
        # We reset the index so that iteration can start from the
beginning.
        self._index = 0
        return self

    def __next__(self):
        # This method returns the next value during iteration.
        if self._index < len(self._attributes):
            result = self._attributes[self._index]
            self._index += 1
            return result
        else:
            # StopIteration is raised when there are no more values to
return
            raise StopIteration

# Testing the class
rectangle = Rectangle(10, 5)

# Iterating over the instance
```

```
for attribute in rectangle:
    print(attribute)
```

Explanation:

1. **`__init__` Method**:
   - Takes `length` and `width` as parameters and stores them as instance variables.
   - Prepares a list of dictionaries (`_attributes`), with one entry for `length` and one for `width`.
2. **`__iter__` Method**:
   - Resets the `_index` to `0` and returns the instance itself, allowing it to be iterated.
3. **`__next__` Method**:
   - Returns the next attribute (either length or width) from the `_attributes` list.
   - Raises `StopIteration` when there are no more items to iterate over.

Output:

When you create a `Rectangle` instance and iterate over it:

```
rectangle = Rectangle(10, 5)
for attribute in rectangle:
    print(attribute)
```

The output will be:

css
Copy code

```
{'length': 10}
{'width': 5}
```

This fulfills the requirements of iterating over the `Rectangle` class to return first the length and then the width in the specified format.