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.

Answer : By default, Django signals are executed synchronously. This means that the signal receiver function will be executed immediately after the signal is sent, within the same thread of execution.

```python
from django.dispatch import Signal

# Create a custom signal
my_signal = Signal()

# Signal receiver function
def my_signal_handler(sender, **kwargs):
    print("Signal received synchronously!")

# Connect the receiver to the signal
my_signal.connect(my_signal_handler)

# Send the signal
my_signal.send(sender=None)
```

Explanation:

1. Create a custom signal: `my_signal` is a custom signal defined using the Signal class.
2. Define a signal receiver: The `my_signal_handler` function is defined as the receiver for the `my_signal`.
3. Connect the receiver: The receiver is connected to the signal using the connect method.
4. Send the signal: The send method is used to send the signal.

When this code is executed, you will see the output "Signal received synchronously!" printed immediately after the signal is sent. This demonstrates that the signal receiver is executed synchronously within the same thread of execution.

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.

Answer : Yes, by default, Django signals are executed in the same thread as the caller. This means that the signal receiver function will run within the same thread of execution as the code that sent the signal.

```python
import threading

from django.dispatch import Signal

# Create a custom signal
my_signal = Signal()

# Signal receiver function
def my_signal_handler(sender, **kwargs):
    print(f"Signal received in thread: {threading.get_ident()}")

# Connect the receiver to the signal
my_signal.connect(my_signal_handler)

# Main thread
def main():
    print(f"Main thread: {threading.get_ident()}")
    my_signal.send(sender=None)

if __name__ == '__main__':
    main()
```

Explanation:

1. Create a signal: We define a custom signal `my_signal`.
2. Define a receiver: The `my_signal_handler` function will be executed when the signal is sent.
3. Connect the receiver: We connect the receiver to the signal.
4. Main thread: The main function runs in the main thread.
5. Send the signal: We send the signal from the main thread.

When you run this code, you'll notice that the output from both the main thread and the signal receiver function will have the same thread ID. This confirms that the signal is executed in the same thread as the caller.

"*While this is the default behavior, there are ways to execute signals asynchronously using task queues like Celery or by manually creating threads. However, for most common use cases, synchronous execution is sufficient.*"

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.

Answer: No, by default, Django signals do not run in the same database transaction as the caller. This means that any changes made within a signal receiver function will not be part of the same transaction that triggered the signal.

```python
from django.db import transaction
from django.dispatch import Signal

# Create a custom signal
my_signal = Signal()

# Model with a transaction
class MyModel(models.Model):
    name = models.CharField(max_length=100)

    def save(self, *args, **kwargs):
        with transaction.atomic():
            super().save(*args, **kwargs)
            my_signal.send(sender=self.__class__)

# Signal receiver function
def my_signal_handler(sender, instance, **kwargs):
    try:
        # Attempt to create a new model instance
        NewModel.objects.create(name="New model")
    except Exception as e:
        print(f"Error creating new model: {e}")

# Connect the receiver to the signal
my_signal.connect(my_signal_handler)

# Create a new model instance
MyModel.objects.create(name="Model instance")
```

Explanation:

1. Create a model with a transaction: The `MyModel` class defines a save method that uses `transaction.atomic()` to ensure that the creation of the model instance is wrapped in a transaction.
2. Send the signal: When a `MyModel` instance is saved, the `my_signal` is sent.
3. Signal receiver function: The `my_signal_handler` attempts to create a new `NewModel` instance within the receiver function.

4. Transaction isolation: If the creation of the new model instance fails due to an error, the changes made within the save method will be rolled back, while the changes made in the signal receiver will not be affected.

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>}`**

Solution :

```python
class Rectangle:
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width

    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

# Example usage:
rect = Rectangle(5, 3)
for attribute in rect:
    print(attribute)
```