Question 1: Are Django signals executed synchronously or asynchronously by default?

As python is a single threader programming language so Django signals are also executed synchronously by default. This means when a signal is sent, Django will immediately run all connected signal handlers before continuing with the rest of the code. There is no delay or background processing .

The output will look something this:
Before sending signal
Signal handler running
After sending signal

This shows that the signal handler runs right away, in between the two print statements, confirming synchronous execution.

Question 2: Do Django signals run in the same thread as the caller?

Yes, Django signals run in the same thread as the code that sends the signal. This means signal handlers don't execute in the background or in a separate thread unless threading or async handling is manually added.

```python
import threading
from django.dispatch import Signal, receiver

my_signal = Signal()


def show_current_thread():
    print("Main thread ID:", threading.get_ident())


@receiver(my_signal)
def my_handler(sender, **kwargs):
    print("Handler thread ID:", threading.get_ident())


show_current_thread()
my_signal.send(sender=None)
```

Both IDs printed will be the same, showing that the signal and the handler execute in the same thread.

Question 3: Do Django signals run in the same database transaction as the caller?

Not necessarily. Signals like post_save are triggered immediately after save() is called, but before the database transaction is committed. This means the changes might not yet be visible to other parts of the code or other database queries. As a result, signal handlers should avoid relying on data that hasn't been fully committed.

```python
from django.db import transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from myapp.models import MyModel


@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    print("Signal handler running")
    print("Autocommit:", transaction.get_autocommit())


with transaction.atomic():
    obj = MyModel.objects.create()
```

In this case, the signal runs inside the transaction block. If something goes wrong and the transaction is rolled back, the signal will have already executed, which may lead to inconsistencies. For cases where the handler must wait until the transaction is fully successful, transaction.on_commit() should be used.