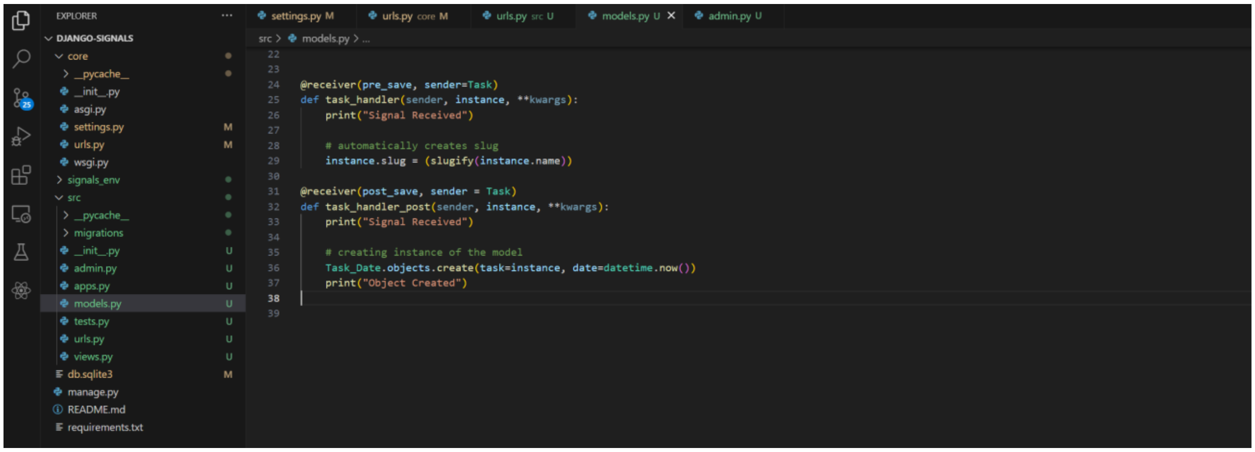
## Django Signals

Django signals allow certain senders to inform a set of receivers that specific actions have occurred. Django signals are used to send and receive specific essential information whenever a data model is saved, changed, or even removed. This relates to specific past or present client-provided events that occur in real time.

**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.

→ By default, when a signal is triggered in Django, all connected receivers are executed in a synchronous manner, meaning that each receiver runs one after the other in a blocking fashion. Django waits for each receiver to complete its execution before moving to the next one, and the entire process halts until all receivers have finished. Developers can also use asynchronous methods in Django allowing the main application to continue running without waiting for the signal receivers to complete.



The pre_save and post_save signals for the Task model are carried out synchronously in the above code. The task_handler method automatically creates a slug from the task name and assigns it to the slug field when a Task instance is saved. This process begins with the pre_save signal. This ensures that the slug is formed in real time. The task_handler_post function is triggered by the post_save signal once the save is finished. It then creates a new Task_Date object with the current date and associates it with the saved Task instance. Because the signal execution is synchronous, the operations must be completed by both handlers before control reverts to the main execution flow, guaranteeing that the sequence of events is followed.

**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.

→ By default, Django signals run in the same thread as the caller. This means that when a signal is triggered, all the connected receivers are executed in the same thread as the operation that triggered the signal. The signal handlers are executed synchronously, meaning the calling code waits for all the signal handlers to complete before continuing. If asynchronous behavior is desired, the signal handling process must be modified to run the signal handlers in a separate thread or process.

```python
class Task(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    slug = models.SlugField(max_length=200, null=True, blank=True)
    is_deleted = models.BooleanField(default=False)

    def __str__(self):
        return self.name

class Task_Date(models.Model):
    task = models.ForeignKey(Task, on_delete=models.CASCADE)
    date = models.DateField()

@receiver(post_save, sender = Task)
def task_handler_post(sender, instance, **kwargs):
    print("Signal Received")

    # simulating a time-consuming task
    # sleep for 5 seconds to simulate delay
    time.sleep(5)
    print("Signal Handler Done")

    # creating instance of the model
    Task_Date.objects.create(task=instance, date=datetime.now())
    print("Object Created")
```

In the provided code, the post_save signal for the Task model is processed synchronously in the same thread as the save operation. When a Task instance is saved, Django triggers the task_handler_post function. This signal handler executes in the same thread as the save operation and performs a time-consuming task by sleeping for 5 seconds, simulating a delay. During this time, the main thread is blocked and cannot proceed until the task_handler_post function completes. Only after the sleep period and the creation of the Task_Date instance does the function print Object Created, indicating the completion of the signal handling. This synchronous execution means that the entire save operation of the Task is delayed by the duration of the signal handler, demonstrating how Django's default behavior of processing signals in the same thread can impact performance and responsiveness.

**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.

→ In Django, signals do not run in the same database transaction as the caller by default. Django's signal framework operates asynchronously with respect to database transactions. When a signal is triggered, it is executed in the context of the current request but not within the same transaction scope as the database operations that caused the signal to fire. This means that changes made by the signal handlers may not be visible within the same transaction if they involve additional database operations.

```python
class Task(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    slug = models.SlugField(max_length=200, null=True, blank=True)
    is_deleted = models.BooleanField(default=False)

    def __str__(self):
        return self.name

class Task_Date(models.Model):
    task = models.ForeignKey(Task, on_delete=models.CASCADE)
    date = models.DateField()

@receiver(post_save, sender = Task)
def task_handler_post(sender, instance, **kwargs):
    print("Signal Received")

    # simulating a time-consuming task
    # sleep for 5 seconds to simulate delay
    time.sleep(5)
    print("Signal Handler Done")

    # ensuring Task_Date creation happens in the same transaction
    with transaction.atomic():
        # creating instance of the model
        Task_Date.objects.create(task=instance, date=datetime.now())
        print("Object Created")
```

In the above code, with transaction.atomic() ensures that the creation of the Task_Date instance is part of the same atomic transaction. This guarantees that the operation is completed only if the transaction succeeds, ensuring consistency. The time.sleep(5) simulates a delay, but the actual creation of the Task_Date object only happens within the atomic transaction, which ensures that the changes are safe and consistent.

**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>}

In [17]:

```python
# creating Rectangle Class

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

    # interating over the instance of class

    def __iter__(self):

        # returning a iterator over the tuple of dictionaries

        return iter ([
            {'length': self.length},
            {'width': self.width}
        ])


for attribute in Rectangle(18,9):
    print(attribute)
```

```
{'length': 18}
{'width': 9}
```