Sending and Receiving

This lesson discusses combining, sending and receiving data in and out of a generator.

Sending & Receiving

Consider the snippet below:

```
def generate_numbers():
    i = 0
    while True:
        i += 1
        yield i
        k = yield
        print(k)
```

You may be surprised how this snippet behaves when we send and receive data.

1. First we create the generator object as follows:

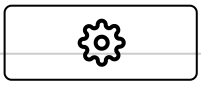
   ```
   generator = generate_numbers()
   ```

   Remember creating the generator object doesn't run the generator function.

2. Next, we start the generator by invoking **`next()`**. We'll receive a value from the generator function since the first **`yield`** statement returns a value. We can do that as follows:

   ```
   item = next(generator)
   ```

```
    print(item)
```



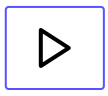
3. It is very important to understand that at this point, the generator's execution is suspended at the first **`yield`** statement. If we try to **`send()`** data, it'll not be received since the generator isn't suspended at a **`yield`** assignment statement. Let's run this scenario so that we understand the concept clearly.

```
def generate_numbers():
    i = 0
    while True:
        i += 1
        yield i
        k = yield
        print(k)


if __name__ == "__main__":
    generator = generate_numbers()

    item = next(generator)
    print(item)

    # Nothing is received by the generator function
    generator.send(5)
```



4. Note that in the above code the generator doesn't receive 5 when we **`send()`** it. The value 5 is lost as the generator isn't suspended at a **`yield`** assignment statement. In fact, the generator resumes execution from the first **`yield`** statement and immediately blocks at the second **`yield`** statement. In between, the two yield

statements no other line of code is executed. The main method which invokes **`send()`** on the generator object receives None because by definition **`send()`** returns the next yielded value in a generator function which is None.

5. We can insert a next or a send to move the generator execution from the first yield to the second yield statement. You can consider this a noop.

6. Once the generator object suspends at the second **`yield`** statement, we can invoke **`send()`** to pass data into the generator function. The generator function would successfully receive the data and at the same time, it'll loop back to the first **`yield`** statement and return the value of **`i`** as the return value of the **`send()`** method. This is demonstrated by the runnable script below:

```
def generate_numbers():
    i = 0
    while True:
        i += 1
        yield i
        k = yield
        print("Received in generator function: " + str(k))


if __name__ == "__main__":
    generator = generate_numbers()

    item = next(generator)
    print("Received in main script: " + str(item))

    # Nothing is received by the generator function
    item = generator.send(5)
    print("Received in main script: " + str(item))

    # The second send is successful
    item = generator.send(5)
    print("Received in main script: " + str(item))
```

7. Note that the generator again suspends itself at the first **`yield`** statement and will require another noop send or next to move to the second **`yield`** statement. The code below adds more statements to send and receive data from the generator function alongwith noop operations.

```
def generate_numbers():
    i = 0
    while True:
        i += 1
        yield i
        k = yield
        print("Received in generator function: " + str(k))


if __name__ == "__main__":
    generator = generate_numbers()

    item = next(generator)
    print("Received in main script: " + str(item))

    # NOOP
    item = next(generator)
    print("Received in main script: " + str(item))

    item = generator.send(55)
    print("Received in main script: " + str(item))

    # NOOP
    item = next(generator)
    print("Received in main script: " + str(item))

    item = generator.send(65)
    print("Received in main script: " + str(item))

    # NOOP
    item = next(generator)
    print("Received in main script: " + str(item))

    item = generator.send(75)
    print("Received in main script: " + str(item))
```

So far we have manually iterated the generator object sending and receiving values from the generator function. We can use a loop, but the caveat is to do the first iteration outside of the loop. The noop operations become part of the loop. For the above example, we can write a loop as follows:

```
    # The first iteration happens outside the loop
    k = next(generator)
    print("Received in main function: " + str(k))


    for i in range(0, 11):
        # The noop operation required to move the generator
        # from the first yield to the second yield statement
        next(generator)

        # send will both pass in the value to the generator
        # function and also yield the next value from the
        # generator
        k = generator.send(i+50)
        print("Received in main function: " + str(k))
```

```
def generate_numbers():
    i = 0
    while True:
        i += 1
        yield i
        k = yield
        print("Received in generator function: " + str(k))


if __name__ == "__main__":
    generator = generate_numbers()

    # The first iteration happens outside the loop
    k = next(generator)
    print("Received in main function: " + str(k))


    for i in range(0, 11):
        # The noop operation required to move the generator
        # from the first yield to the second yield statement
        next(generator)

        # send will both pass in the value to the generator
        # function and also yield the next value from the
        # generator
        k = generator.send(i+50)
        print("Received in main function: " + str(k))
```

Back

Next

Send

... continued

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/sending-and-receiving__asyncio__python-concurrency-for-senior-engineering-interviews)