... continued

Continues the discussion on sending and receiving data from generators.

Before we end our discussion on generators and the **`yield`** statement, let's see another way it can be used. In the example from the previous section, we used two **`yield`** statements to send and receive data in and out of the generator. We can instead use a single one to do both and without the need to do noop operations. The generator method to do this appears below:

```
def generate_numbers():
    i = 0

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

In the above snippet, the following statement is of interest to us:

```
k = (yield i)
```

It may appear confusing at first look but consider the statement as:

```
k = <expression>
```

The variable **`k`** gets assigned an expression. The expression itself is **`yield i`** so the generator will first yield a value back to the caller and then receive a value from the caller and assign it to the variable **`k`**.

Given our understanding, we can start the generator using either next or by sending None. Note that if we send None, the value is lost and not received by the generator function because the expression gets evaluated first which returns a value.

Once the generator is started, we can use a for loop to send and receive values from the generator function, without inserting any noop operations to resume execution from one yield to another as we did in the previous section's example. The for loop would look as follows:

```
    for i in range(0, 5):
        item = generator.send(55 + i)
        print("received " + str(item))
```

The complete code appears below:

```
def generate_numbers():
    i = 0

    while True:
        i += 1
        k = (yield i)
        print(k)


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

    item = generator.send(None)
    print("received " + str(item))

    for i in range(0, 5):
        item = generator.send(55 + i)
        print("received " + str(item))
```

The above code appears as runnable script in the code widget below.

```
def generate_numbers():
    i = 0

    while True:
        i += 1
        k = (yield i)
        print(k)


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

    item = generator.send(None)
    print("received " + str(item))

    for i in range(0, 5):
        item = generator.send(55 + i)
        print("received " + str(item))
```

Send or Receive or both?

Generally, you'll see that generator functions do one of the two but not both. The same advice: to not mix iteration and sending is expressed by the influential Pythonista Dave Beazley in his tutorial here (https://www.youtube.com/watch?v=Z_OAlIhXziw&feature=youtu.be&t=1820). Use generators to generate values and coroutines to consume values. Generator functions receive values are called coroutines. We'll study them next.

Sending and Receiving

Mark as Completed

Report an Issue

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