Yield From

This lesson explains the yield from syntax.

Yield From

The **`yield from`** syntax was introduced in PEP-380. The stated motivation for introducing the syntax is to enable refactoring of existing code that utilizes **`yield`** easier. The yield from syntax is as follows:

```
yield from <expr>
```

The expression must be an iterable from which an iterator is extracted. Let's understand the problem that **`yield from`** solves. Consider the following snippet of code:

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i


def outer_generator():
    nested_gen = nested_generator()

    for item in nested_gen:
        yield item

if __name__ == "__main__":

    gen = outer_generator()

    for item in gen:
        print(item)
```

The above code has two generator functions. The **`outer_generator()`** calls the **`nested_generator()`** in a loop and returns values from the inner generator to the main script. You can run the code below and examine the output.

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i


def outer_generator():
    nested_gen = nested_generator()

    for item in nested_gen:
        yield item

if __name__ == "__main__":

    gen = outer_generator()

    for item in gen:
        print(item)
```

We can refactor the above code and remove the *for* loop in the **`outer_generator()`** as follows, with the same output.

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i

def outer_generator_with_yield_from():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen_using_yield_from = outer_generator_with_yield_from()

    for item in gen_using_yield_from:
        print(item)
```

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i

def outer_generator_with_yield_from():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen_using_yield_from = outer_generator_with_yield_from()

    for item in gen_using_yield_from:
        print(item)
```

The **`yield from`** expects an iterable on its right and runs it to exhaustion. Remember a generator is after all an iterator! In fact the following test for instance-of will return true.

```
isinstance(nested_gen, collections.abc.Iterable)
```

In this example the **`outer_generator()`** is called the **delegating generator** and the **`nested generator`** is called the **subgenerator**. However, it would seem rather silly to introduce new syntax in the language just to rid oneself of an extra loop. In fact, **`yield from`** is much more powerful. Remember that PEP-342 endowed simple Python generators with additional methods transforming them into coroutines. The code required to correctly work with subgenerators that use these additional methods becomes much more complex and **`yield from`** hides all that complexity from the developer.

Yield from Using Send

Things get more interesting when we throw in the **`send()`** method. Consider the snippet below:

```
def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        # receive the value from the caller
        k = yield
        try:
            # send the value to the inner generator
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            gen.send(i)
        except StopIteration:
            pass
```

```
def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        # receive the value from the caller
        k = yield
        try:
            # send the value to the inner generator
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            gen.send(i)
        except StopIteration:
            pass
```

Not only is the code verbose but confusing to understand. The outer generator is acting as an intermediary to pass the values it receives from the caller to the inner generator. The above monstrosity can be simplified as follows:

```
def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            gen.send(i)
        except StopIteration:
            pass
```

```
def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            gen.send(i)
    except StopIteration:
        pass
```

With **`send()`** the output of the code doesn't change before and after **`yield from`**. However, if the calling script wanted to throw an exception to the inner generator the output would be unexpected. Consider the following snippet:

```
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        k = yield
        try:
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("delibrate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

In the above snippet, the main caller tries to throw an exception to the inner generator however the intermediary generator receives and the program halts because the outer generator doesn't handle exceptions.

```
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        k = yield
        try:
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("delibrate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

Of course, we can handle the situation by adding another **`try-except`** to cater for the exception the main thread throws, catching and forwarding it to the inner generator. However, as you can see, the code would become more verbose and complex. The fix for the above code without **`yield from`** would look like:

```
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        try:
            k = yield
        except Exception as e:
            nested_gen.throw(e)
        try:
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("deliberate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

```
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    next(nested_gen)

    for _ in range(5):
        try:
            k = yield
        except Exception as e:
            nested_gen.throw(e)
        try:
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("delibrate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

Examine the output of the above program and notice the program doesn't crash this time around. You may now begin to appreciate the simplicity that **`yield from`** brings. We can simplify the above using **`yield from`** as follows:

```python
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("deliberate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

```python
def nested_generator():
    for _ in range(5):
        try:
            k = yield
            print("inner generator received = " + str(k))
        except Exception:
            print("caught an exception")


def outer_generator():
    nested_gen = nested_generator()
    yield from nested_gen


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    for i in range(5):
        try:
            if i == 1:
                gen.throw(Exception("deliberate exception"))
            else:
                gen.send(i)
        except StopIteration:
            pass
```

Yield from with close

Without using **`yield from`** if we execute **`close()`** on the outer generator the inner generator will be left suspended. Run the code below and examine the output:

```
import inspect

var = None


def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    global var
    nested_gen = nested_generator()
    var = nested_gen
    next(nested_gen)

    for _ in range(5):
        k = yield
        try:
            nested_gen.send(k)
        except StopIteration:
            pass


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    try:
        gen.close()
        print("Outer generator state: " + inspect.getgeneratorstate(gen))
        print("Inner generator state: " + inspect.getgeneratorstate(var))

    except StopIteration:
        pass
```

Contrast the above output with the output we get when we use **`yield from`**. Both the generators are closed.

```
import inspect

var = None


def nested_generator():
    for _ in range(5):
        k = yield
        print("inner generator received = " + str(k))


def outer_generator():
    global var
    nested_gen = nested_generator()
    var = nested_gen
    yield from nested_gen


if __name__ == "__main__":

    gen = outer_generator()
    next(gen)

    try:
        gen.close()
        print("Outer generator state: " + inspect.getgeneratorstate(gen))
        print("Inner generator state: " + inspect.getgeneratorstate(var))

    except StopIteration:
        pass
```

`yield from` can be best thought of as **creating transparent bidirectional communication between the caller and the subgenerator.**

Back

Next

... continued

... continued

Mark as Completed

Report an Issue

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