Yield

This lesson discusses the yield keyword and its uses.

Yield

The word **yield** is defined as to produce or provide or to *give way to arguments or pressure.* When you are on the road you may come across "yield to pedestrians" sign boards which require you to stop and give way to pedestrians crossing the road. Python's use of the word yield can both produce values and also give way as we shortly explain.

Consider the program below, which returns a string.

```
def keep_learning_synchronous():
    return "Educative"
```

We can invoke this function simply as:

```
    str = keep_learning_synchronous()
    print(str)
```

The above program is synchronous, it completes executing the method `keep_learning_synchronous()` and then the control returns to the invoking script and the print statement is executed. The code above appears below in the runnable widget.

```
def keep_learning_synchronous():
    return "Educative"


if __name__ == "__main__":
    str = keep_learning_synchronous()
    print(str)
```

Now let's replace the **`return`** statement with **`yield`** and see what happens. The changes are:

```
def keep_learning_asynchronous():
    yield "Educative"


if __name__ == "__main__":
    str = keep_learning_asynchronous()
    print(str)
```

Observe the output of the above snippet and how it differs from the previous one.

```
def keep_learning_asynchronous():
    yield "Educative"


if __name__ == "__main__":
    str = keep_learning_asynchronous()
    print(str)
```

So instead of being returned the string, we are returned a **generator object** as the print statement shows. In fact our method `keep_learning_asynchronous()` is now a generator function. Generator functions are called as generators because they *generate* values. We discuss generator in more depth later but for now consider them as iterators. In order for the generator object to produce or yield us the string we invoke `next()` on it. The changes are:

```
    gen = keep_learning_asynchronous()
    str = next(gen)
    print(str)
```

```
def keep_learning_asynchronous():
    yield "Educative"


if __name__ == "__main__":
    gen = keep_learning_asynchronous()
    str = next(gen)
    print(str)
```

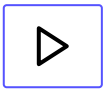
Since we are passing generator objects into `next()` we can conclude that generators are **iterators**. We can also use them with for loops or other iteration tools. Let's rewrite our program with a for loop now.

Using a for loop with a generator

```
    gen = keep_learning_asynchronous()
    for item in gen:
```

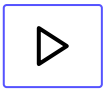
```
        print(item)
```


```
def keep_learning_asynchronous():
    yield "Educative"


if __name__ == "__main__":
    gen = keep_learning_asynchronous()
    for item in gen:
        print(item)
```




We can add multiple yield statements in the same program and retrieve invoking **`next()`** on the returned generator object. For instance in the snippet below, we have two yield statements in the **`keep_learning_asynchronous()`** method.

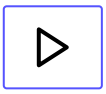
```
def keep_learning_asynchronous():
    yield "Educative"
    yield "is great!"


if __name__ == "__main__":
    gen = keep_learning_asynchronous()
    for item in gen:
        print(item)
```




If we attempt to invoke **`next()`** on a generator object that had already produced (yielded) all its values, we'll be thrown a **`StopIteration`** exception. Consider the same program above but now we invoke next once the for loop ends. Observe the exception from running the snippet

below:


```
def keep_learning_asynchronous():
    yield "Educative"
    yield "is great!"


if __name__ == "__main__":
    gen = keep_learning_asynchronous()
    for item in gen:
        print(item)

    # Throws StopIteration exception
    next(gen)
```




Return and Yield

You may be tempted to wonder if we can combine **`return`** and **`yield`** in the same generator function. In fact we can, consider the snippet below:

```
def keep_learning_asynchronous():
    yield "Educative"
    return "is great"
```

We know that we can invoke **`next()`** on the generator object returned from this method and get the first string. If we invoke **`next()`** a second time we will receive a **`StopIteration`** exception. The second string will be passed in as the value to **`StopIteration`** exception. We can access the second string from the exception object as follows:

```
    gen = keep_learning_asynchronous()

    first_string = next(gen)
    print(first_string)

    try:
        next(gen)
    except StopIteration as e:
        second_string = e.value
        print(second_string)
```

```
def keep_learning_asynchronous():
    yield "Educative"
    return "is great"


if __name__ == "__main__":
    gen = keep_learning_asynchronous()

    first_string = next(gen)
    print(first_string)

    try:
        next(gen)
    except StopIteration as e:
        second_string = e.value
        print(second_string)
```

Anytime you **`return`** from a generator function, it'll be equivalent of raising the **`StopIteration`** exception.

Suspending State

The true power of **`yield`** is unleashed when we want to return a sequence or a list from a method but don't want to compute the entire sequence up front. Consider a situation where you want to grab prime numbers in order. However, the logic of your program can't determine beforehand how many you will need. It would be unwise to precompute the first million prime numbers especially if you are not using a sophisticated algorithm to determine the primality of a number.

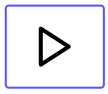
When the **`yield`** statement is encountered the state of the function is suspended and the value being yielded is returned to the caller. At this point enough state is saved to resume the generator function on a subsequent **`next()`**.

Let's write a function that prints prime numbers and then we'll convert it into a generator. Our method would look something like as following:

```
def get_primes():
    i = 1
    while True:
        if is_prime(i):
            print(i)
        i += 1
```

Don't fret about the **`is_prime()`**. Assume it is correctly implemented and returns a boolean to indicate whether a number is prime or not. Also, at this point, we aren't concerned with the efficiency of the algorithm. The complete code appears in the runnable widget. Note that we tweak the method to print only the primes till hundred, else the widget will timeout.


```
def is_prime(i):
    if i == 1 or i == 2 or i == 3:
        return True

    k = 2

    while k <= i / 2:
        if i % k == 0:
            return False
        k += 1

    return True


def get_primes():
    i = 1
    while i < 100:
        if is_prime(i):
            print(i)
        i += 1


if __name__ == "__main__":
    get_primes()
```




Now our task is to turn the original method into a generator function. We simply replace the print statement with a **`yield`** as follows.

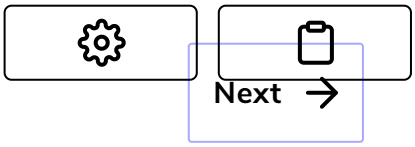
```
def get_primes():
    i = 1
    while True:
        if is_prime(i):
            yield i
        i += 1
```

We can now retain the infinite loop and work the iterator returned by the **`get_primes()`** method to get prime numbers. In the runnable snippet below we are printing five primes using **`next()`**.

```
def is_prime(i):
    if i == 1 or i == 2 or i == 3:
        return True

    k = 2

    while k <= i / 2:
        if i % k == 0:
            return False
        k += 1

    return True


def get_primes():
    i = 1
    while True:
        if is_prime(i):
            yield i
        i += 1


if __name__ == "__main__":
    gen = get_primes()

    print(next(gen))
    print(next(gen))
    print(next(gen))
    print(next(gen))
    print(next(gen))
```

In summary, we can use yield in a function as **`yield <expression>`**. Yield allows a function to return a value and let the state of the function suspend till **`next()`** is invoked on the associated generator object.

This lesson should provide a solid grounding in how the **`yield`** keyword is used and prepare you to get into more depth with generator functions.



Back

Next

Iterables

Generator

Mark as Completed

Report an Issue

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