YIELDING AND GENERATORS

Iterators review

Let's recall how we would write a simple iterator for factorials

```
class FactIter:
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= self.n:
            raise StopIteration
        else:
            result = math.factorial(self.i)
            self.i += 1
            return result
```

Now that's quite a bit of work for a simple iterator!

There has to be a better way...

What if we could do something like this instead:

```
def factorials(n):
    for i in range(n):
        emit factorial(i)
        pause execution here
        wait for resume
    return 'done!'
```

and in our code we would want to do something like this maybe:

```
facts = factorials(4)
get_next(facts)  → 0!
get_next(facts)  → 1!
get_next(facts)  → 2!
get_next(facts)  → 3!
get_next(facts)  → done!
```

Of course, getting `0!`, `1!`, `2!`, `3!` followed by a string is odd

And what happens if we call `get_next` again?

Maybe we should consider raising an exception... `StopIteration`?

And instead of calling `get_next`, why not just use `next`?

But what about that `emit`, `pause`, `resume`? → `yield`

Yield to the rescue...

The `yield` keyword does exactly what we want:

- it emits a value
- the function is effectively suspended (but it retains its current state)
- calling `next` on the function resumes running the function right after the yield statement
- if function `returns` something instead of yielding (finishes running) → `StopIteration` exception

```
def song():
    print('line 1')
    yield "I'm a lumberjack and I'm OK"
    print('line 2')
    yield 'I sleep all night and I work all day'
```

`lines = song()` → no output!

`line = next(lines)` → `'line 1'` is printed in console
`line` → "I'm a lumberjack and I'm OK"

`line = next(lines)` → `'line 2'` is printed in console
`line` → "I sleep all night and I work all day"

`line = next(lines)` → `StopIteration`

Generators

A function that uses the `yield` statement, is called a generator function

```
def my_func():
    yield 1
    yield 2
    yield 3
```

`my_func` is just a regular function

calling my_func() returns a generator object

We can think of functions that contain the `yield` statement as generator factories

The generator is created by Python when the function is called → `gen = my_func()`

The resulting generator is executed by calling `next()` → `next(gen)`

the function body will execute until it encounters a `yield` statement

it yields the value (as return value of `next()`) then it suspends itself

until `next` is called again → suspended function resumes execution

if it encounters a `return` before a `yield`

→ `StopIteration` exception occurs

(Remember that if a function terminates without an explicit return, Python essentially returns a None value for us)

Generators

```
def my_func():
    yield 1
    yield 2
    yield 3

gen = my_func()
```

→ gen is a generator

```
next(gen)
```

→ 1

```
next(gen)
```

→ 2

```
next(gen)
```

→ 3

```
next(gen)
```

→ StopIteration

Generators

`next` `StopIteration`

This should remind you of iterators!

In fact, generators are iterators → they implement the iterator protocol

`__iter__` `__next__`

```
def my_func():
    yield 1
    yield 2
    yield 3

gen = my_func()
```

→ they are exhausted when function returns a value

→ `StopIteration` exception

→ return value is the exception message

`gen.__iter__()` → `iter(gen)` → returns `gen` itself

`gen.__next__()` → `next(gen)`

Example

```
class FactIter:
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= self.n:
            raise StopIteration
        else:
            result = math.factorial(self.i)
            self.i += 1
            return result

fact_iter = FactIter(5)
```

```
def factorials(n):
    for i in range(n):
        yield math.factorial(i)

fact_iter = factorials(5)
```

Generators

Generator functions are functions which contain at least one yield statement

When a generator function is called, Python creates a generator object

Generators implement the iterator protocol

Generators are inherently lazy iterators (and can be infinite)

Generators are iterators, and can be used in the same way (for loops, comprehensions, etc)

Generators become exhausted once the function returns a value

Code Exercises