A Fibonacci Iterator

Now you’re ready to learn how to build an iterator. An iterator is just a class that defines an `__iter__()` method.

```
class Fib:                                        #①
    def __init__(self, max):                      #②
        self.max = max

    def __iter__(self):                           #③
        self.a = 0
        self.b = 1
        return self

    def __next__(self):                           #④
        fib = self.a
        if fib > self.max:
            raise StopIteration                   #⑤
        self.a, self.b = self.b, self.a + self.b
        return fib                                #⑥
```

① To build an iterator from scratch, `Fib` needs to be a class, not a function.

② “Calling” `Fib(max)` is really creating an instance of this class and calling its `__init__()` method with `max`. The `__init__()` method saves the maximum value as an instance variable so other methods can refer to it later.

③ The `__iter__(`) method is called whenever someone calls `iter(fib)`. (As you’ll see in a minute, a `for` loop will call this automatically, but you can also call it yourself manually.) After performing beginning-of-iteration initialization (in this case, resetting `self.a` and `self.b`, our two counters), the `__iter__()` method can return any object that implements a `__next__()` method. In this case (and in most cases), `__iter__()` simply returns `self`, since this class implements its own `__next__()` method.

④ The `__next__()` method is called whenever someone calls `next()` on an iterator of an instance of a class. That will make more sense in a minute.

⑤ When the `__next__()` method raises a `StopIteration` exception, this signals

⑤ When the `__next__()` method raises a `StopIteration` exception, this signals to the caller that the iteration is exhausted. Unlike most exceptions, this is not an error; it's a normal condition that just means that the iterator has no more values to generate. If the caller is a `for` loop, it will notice this `StopIteration` exception and gracefully exit the loop. (In other words, it will swallow the exception.) This little bit of magic is actually the key to using iterators in `for` loops.

⑥ To spit out the next value, an iterator's `__next__()` method simply `returns` the value. Do not use `yield` here; that's a bit of syntactic sugar that only applies when you're using generators. Here you're creating your own iterator from scratch; use `return` instead.

> *All three of these class methods,* `__init__`*,* `__iter__`*, and* `__next__`*, begin and end with a pair of underscore (* `_` *) characters. Why is that? There's nothing magical about it, but it usually indicates that these are "special methods." The only thing "special" about special methods is that they aren't called directly; Python calls them when you use some other syntax on the class or an instance of the class. More about special methods.*

Thoroughly confused yet? Excellent. Let's see how to call this iterator:

```
from fibonacci2 import Fib
for n in Fib(1000):
     print(n, end=' ')
#0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Why, it's exactly the same! Byte for byte identical to how you called Fibonacci-as-a-generator (modulo one capital letter). But how?

There's a bit of magic involved in `for` loops. Here's what happens:

- The `for` loop calls `Fib(1000)`, as shown. This returns an instance of the `Fib` class. Call this `fib_inst`.
- Secretly, and quite cleverly, the `for` loop calls `iter(fib_inst)`, which returns an iterator object. Call this `fib_iter`. In this case, `fib_iter == fib_inst`, because the `__iter__()` method returns `self`, but the `for` loop

`fib_inst`, because the `__iter__()` method returns `self`, but the `for` loop doesn’t know (or care) about that.

- To “loop through” the iterator, the `for` loop calls `next(fib_iter)`, which calls the `__next__()` method on the `fib_iter` object, which does the next-Fibonacci-number calculations and returns a value. The `for` loop takes this value and assigns it to `n`, then executes the body of the `for` loop for that value of `n`.
- How does the `for` loop know when to stop? I’m glad you asked! When `next(fib_iter)` raises a `StopIteration` exception, the `for` loop will swallow the exception and gracefully exit. (Any other exception will pass through and be raised as usual.) And where have you seen a `StopIteration` exception? In the `__next__()` method, of course!