Part 4: Generators

Bookmark this page

Part 4: Generators

Generators give you an iterator object with no access to the underlying data … if it even exists. Conceptually, iterators are about various ways to loop over data. They can generate data on the fly. In general, you can use either an iterator or a generator — in fact, a generator is a type of iterator. Generators do some of the book-keeping for you and therefore involve simpler syntax.

Yield

```
def a_generator_function(params):
    some_stuff
    yield something
```

Generator functions “yield” a value, rather than returning a value. It *does* ‘return’ a value, but rather than ending execution of the function, it preserves function state so that it can pick up where it left off. In other words, state is preserved between yields.

A function with `yield` in it is a factory for a generator. Each time you call it, you get a new generator:

```
gen_a = a_generator()
gen_b = a_generator()
```

Each instance keeps its own state.

To master yield, you must understand that when you call the function, the code you have written in the function body does not run. The function only returns the generator object. The actual code in the function is run when next() is called on the generator itself.

An example: an implementation of range() as a generator:

```
def y_range(start, stop, step=1):
    i = start
    while i < stop:
        yield i
        i += step
```

Generator Comprehensions: yet another way to make a generator:

```
>>> [x * 2 for x in [1, 2, 3]]
[2, 4, 6]
>>> (x * 2 for x in [1, 2, 3])
<generator object <genexpr> at 0x10911bf50>
>>> for n in (x * 2 for x in [1, 2, 3]):
...   print n
```

‹ Previous Next ›