# Generators

WE'LL COVER THE FOLLOWING

- A Fibonacci Generator
- A Plural Rule Generator

Wouldn't it be grand to have a generic `plural()` function that parses the rules file? Get rules, check for a match, apply appropriate transformation, go to next rule. That's all the `plural()` function has to do, and that's all the `plural()` function should do.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
            yield build_match_and_apply_functions(pattern, search, replace)

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

How the heck does *that* work? Let's look at an interactive example first.

```
def make_counter(x):
     print('entering make_counter')
     while True:
         yield x                          #①
         print('incrementing x')
         x = x + 1


counter = make_counter(2)                 #②
print (counter)                           #③
#<generator object at 0x001C9C10>

print (next(counter))                     #④
#entering make_counter
```

```
#2

print (next(counter))                    #⑤

#incrementing x
#3

print (next(counter))                    #⑥
#incrementing x
#4
```

① The presence of the `yield` keyword in `make_counter` means that this is not a normal function. It is a special kind of function which generates values one at a time. You can think of it as a resumable function. Calling it will return a *generator* that can be used to generate successive values of `x`.

② To create an instance of the `make_counter` generator, just call it like any other function. Note that this does not actually execute the function code. You can tell this because the first line of the `make_counter()` function calls `print()`, but nothing has been printed yet.

③ The `make_counter()` function returns a generator object.

④ The `next()` function takes a generator object and returns its next value. The first time you call `next()` with the counter generator, it executes the code in `make_counter()` up to the first yield statement, then returns the value that was yielded. In this case, that will be `2`, because you originally created the generator by calling `make_counter(2)`.

⑤ Repeatedly calling `next()` with the same generator object resumes exactly where it left off and continues until it hits the next yield statement. All variables, local state, &c. are saved on yield and restored on `next()`. The next line of code waiting to be executed calls `print()`, which prints incrementing `x`. After that, the statement `x = x + 1`. Then it loops through the `while` loop again, and the first thing it hits is the statement yield `x`, which saves the state of everything and returns the current value of `x` (now `3`).

⑥ The second time you call `next(counter)`, you do all the same things again, but this time `x` is now `4`.

Since `make_counter` sets up an infinite loop, you could theoretically do this forever, and it would just keep incrementing `x` and spitting out values. But

forever, and it would just keep incrementing `x` and spitting out values. But let's look at more productive uses of generators instead.

## A Fibonacci Generator #

```
def fib(max):
    a, b = 0, 1           #①
    while a < max:
        yield a           #②
        a, b = b, a + b   #③
```

① The Fibonacci sequence is a sequence of numbers where each number is the sum of the two numbers before it. It starts with 0 and 1, goes up slowly at first, then more and more rapidly. To start the sequence, you need two variables: a starts at 0, and b starts at 1.

② `a` is the current number in the sequence, so yield it.

③ `b` is the next number in the sequence, so assign that to `a`, but also calculate the next value `(a + b)` and assign that to `b` for later use. Note that this happens in parallel; if `a` is `3` and `b` is `5`, then `a, b = b, a + b` will set `a` to `5` (the previous value of `b`) and `b` to `8` (the sum of the previous values of `a` and `b`).

"yield" pauses a function. "next()" resumes where it left off.

So you have a function that spits out successive Fibonacci numbers. Sure, you could do that with recursion, but this way is easier to read. Also, it works well with `for` loops.

```
def fib(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a + b

for n in fib(1000):       #①
    print(n, end=' ')     #②

#0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

print (list(fib(1000)))  #③
#[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

① You can use a generator like `fib()` in a `for` loop directly. The `for` loop will automatically call the `next()` function to get values from the `fib()` generator and assign them to the `for` loop index variable (`n`).

② Each time through the `for` loop, `n` gets a new value from the yield statement in `fib()`, and all you have to do is print it out. Once `fib()` runs out of numbers (a becomes bigger than max, which in this case is `1000`), then the `for` loop exits gracefully.

③ This is a useful idiom: pass a generator to the `list()` function, and it will iterate through the entire generator (just like the `for` loop in the previous example) and return a list of all the values.

## A Plural Rule Generator #

Let's go back to `plural5.py` and see how this version of the `plural()` function works.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)                    #①
            yield build_match_and_apply_functions(pattern, search, replace)  #②

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):                    #③
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

① No magic here. Remember that the lines of the rules file have three values separated by whitespace, so you use `line.split(None, 3)` to get the three "columns" and assign them to three local variables.

② *And then you yield.* What do you yield? Two functions, built dynamically with your old friend, `build_match_and_apply_functions()`, which is identical to the previous examples. In other words, `rules()` is a generator that spits out match and apply functions on *demand*.

③ Since `rules()` is a generator, you can use it directly in a `for` loop. The first time through the `for` loop, you will call the `rules()` function, which will open

the pattern file, read the first line, dynamically build a match function and an apply function from the patterns on that line, and yield the dynamically built functions. The second time through the `for` loop, you will pick up exactly where you left off in `rules()` (which was in the middle of the for line in `pattern_file` loop). The first thing it will do is read the next line of the file (which is still open), dynamically build another match and apply function based on the patterns on that line in the file, and yield the two functions.

What have you gained over stage 4? Startup time. In stage 4, when you imported the `plural4` module, it read the entire patterns file and built a list of all the possible rules, before you could even think about calling the `plural()` function. With generators, you can do everything lazily: you read the first rule and create functions and try them, and if that works you don't ever read the rest of the file or create any other functions.

What have you lost? Performance! Every time you call the `plural()` function, the `rules()` generator starts over from the beginning — which means re-opening the patterns file and reading from the beginning, one line at a time.

What if you could have the best of both worlds: minimal startup cost (don't execute any code on `import`), *and* maximum performance (don't build the same functions over and over again). Oh, and you still want to keep the rules in a separate file (because code is code and data is data), just as long as you never have to read the same line twice.

To do that, you'll need to build your own iterator. But before you do that, you need to learn about Python classes.