Round and Round - Using Generators in Python

Author	josh
Date	2018-06-23 23:33:06

Overview

Most all modern programming languages support constructs for storing lists of data - think C++ arrays, Java ArrayLists, and Python lists. Any time we have a list of data, it's often necessary to use loops to perform some operation on each item in that list. Again, most modern languages support ways of looping over these lists of data, most commonly "for" loops.

In Python, it's trivial to iterate over a list using a `for` loop. However, Python lists are stored in memory. What happens if the data set you need to operate on is large and therefore memory inefficient? Python offers constructs called iterators that allow one to loop over data sets that are not stored in memory. Some iterators are built in for things like file reading, but you can easily create your own custom iterators using a concept called generators!

Iterators? Generators? Combobulators?

Traditional List Iteration

It's easy to create a list of numbers in Python an loop over that list. Let's say for example, we want to create a list of multiples of 2. We'll store 5 numbers in this list. For each number in this list, we'll just print it out to the screen for now:

```
def print_multiples():

    multiples = get_multiples()
    for m in multiples:
        print m

def get_multiples():

    return [ 2, 4, 6, 8, 10 ]

if __name__ == "__main__":

    print_multiples()
```

Let's step through this code a bit and explain how it works. When we enter the `print_multiples` function from main, the first call `multiples = get_multiples()` assigns the return value of `get_multiples()` to the multiples variable.

The `multiples` variable now stores an *in-memory* list with all of our multiple of 2 values - 2, 4, 6, 8, and 10. We next go to the for loop `for m in multiples:`. These Python for loops are slick and fairly straightforward - for each go around the loop, the next value in the list is stored directly in the variable `m`. The loops continues until each value in the list is exhausted.

The output looks like this:

```
python test.py
2
4
6
8
10
```

So what's a generator look like?

Now let's try this code again, using Python's generator construct. Here's what that looks like:

```
def print_multiples():

    multiples = get_multiples()
    for m in multiples:
        print m

def get_multiples():

    for i in range(1, 6):
        yield i * 2

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

Notice our output is the same:

```
python test.py
2
4
6
8
10
```

This code requires some closer examination to understand how it works. When we assign `multiples = get_multiples()`, we don't a assign a list, we assign what's known in Python as an *iterator*. An iterator object exposes a `next` method that allows for loops to retrieve each sequential item in a list or other iterable object.

When we enter our for loop this time, we don't iterate over the list - instead our iterator uses the `get_multiples()` *generator* function to retrieve each item one by one. The first time we go around the for loop in `print_multiples`, we enter the `get_multiples` function and enter *its* for loop.

Now, you'll notice a different keyword being using in `get_multiples` - instead of *returning* an entire list, the function only *yields* one item, the result of `i * 2`. The value is passed through the iterator's `next` function and printed to the screen. The next time around the for loop in `print_multiples`, the code goes to the same spot the value was yielded from in `get_multiples`. The `return` keyword *returns* code control to the caller, whereas the `yield` keyword only *temporarily yields* control back to the caller until the next time the caller needs a

value from the iterator. The `get_multiples` function's for loop continues, and yields the next multiple of 2. The function will continue yielding values of 2 until it's for loop ends, yielding 10.

Cool, so why would I want to use generators instead of a list?

Our trivial example makes the use of generators pretty clear, but it doesn't explain why they're actually useful. Why all the extra complexity to avoid storing a whole 5 numbers in memory?. The use case of generators goes far beyond small lists of numbers.

First, what if we wanted to display the first one *billion* multiples of 2? In that case, it becomes much more expensive to store one billion integers in memory - it would eat up over a gigagbyte!.

In real-world software engineering applications, we often use generators to deal with large datasets even beyond simple calculations such as this. Generators can be used to operate on large database queries or data read from files on disk, where it would be inefficient or even impossible to read the information into memory.

Generate your own generators

In this article, we've explained how to go beyond memory-stored lists of information and create our own generators. Instead of hogging memory for large data operations, we can make our own memory-efficient iterators. Now when you have large calculations, database queries, or file reads to worry about, you can keep your memory usage low and your code easy to understand thanks to Python!