5. Iterators & Generators

5.1. Iterators

We use `for` statement for looping over a list.

```
>>> for i in [1, 2, 3, 4]:
...     print i,
...
1
2
3
4
```

If we use it with a string, it loops over its characters.

```
>>> for c in "python":
...     print c
...
p
y
t
h
o
n
```

If we use it with a dictionary, it loops over its keys.

```
>>> for k in {"x": 1, "y": 2}:
...     print k
...
y
x
```

If we use it with a file, it loops over lines of the file.

```
>>> for line in open("a.txt"):
...     print line,
...
first line
second line
```

So there are many types of objects which can be used with a for loop. These are called iterable objects.

There are many functions which consume these iterables.

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

5.1.1. The Iteraton Protocol

The built-in function `iter` takes an iterable object and returns an iterator.

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Each time we call the `next` method on the iterator gives us the next element. If there are no more elements, it raises a *StopIteration*.

Iterators are implemented as classes. Here is an iterator that works like built-in `xrange` function.

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

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

The `__iter__` method is what makes an object iterable. Behind the scenes, the *iter* function calls `__iter__` method on the given object.

The return value of `__iter__` is an iterator. It should have a `next` method and raise `StopIteration` when there are no more elements.

Lets try it out:

```
>>> y = yrange(3)
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in next
StopIteration
```

Many built-in functions accept iterators as arguments.

```
>>> list(yrange(5))
[0, 1, 2, 3, 4]
>>> sum(yrange(5))
10
```

In the above case, both the iterable and iterator are the same object. Notice that the `__iter__` method returned `self`. It need not be the case always.

```
class zrange:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return zrange_iter(self.n)

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

    def __iter__(self):
        # Iterators are iterables too.
        # Adding this functions to make them so.
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

If both iteratable and iterator are the same object, it is consumed in a single iteration.

```
>>> y = yrange(5)
>>> list(y)
[0, 1, 2, 3, 4]
>>> list(y)
[]
>>> z = zrange(5)
>>> list(z)
[0, 1, 2, 3, 4]
>>> list(z)
[0, 1, 2, 3, 4]
```

Problem 1: Write an iterator class `reverse_iter`, that takes a list and iterates it from the reverse direction. ::

```
>>> it = reverse_iter([1, 2, 3, 4])
>>> it.next()
4
>>> it.next()
3
>>> it.next()
2
>>> it.next()
1
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

5.2. Generators

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Each time the `yield` statement is executed the function generates a new value.

```
>>> y = yrange(3)
>>> y
<generator object yrange at 0x401f30>
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

So a generator is also an iterator. You don't have to worry about the iterator protocol.

The word "generator" is confusingly used to mean both the function that generates and what it generates. In this chapter, I'll use the word "generator" to mean the genearted object and "generator function" to mean the function that generates it.

Can you think about how it is working internally?

When a generator function is called, it returns an generator object without even beginning execution of the function. When *next*`method is called for the first time, the function starts executing until it reaches `yield` statement. The yielded value is returned by the `next` call.

The following example demonstrates the interplay between `yield` and call to `next` method on generator object.

```
>>> def foo():
...     print "begin"
...     for i in range(3):
...         print "before yield", i
...         yield i
...         print "after yield", i
...     print "end"
...
>>> f = foo()
>>> f.next()
begin
before yield 0
0
>>> f.next()
after yield 0
before yield 1
1
>>> f.next()
after yield 1
before yield 2
2
>>> f.next()
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Lets see an example:

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(seq.next())
    except StopIteration:
        pass
    return result

print take(5, squares()) # prints [1, 4, 9, 16, 25]
```

5.3. Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

We can use the generator expressions as arguments to various functions that consume iterators.

```
>>> sum(((x*x for x in range(10)))
285
```

When there is only one argument to the calling function, the parenthesis around generator expression can be omitted.

```
>>> sum(x*x for x in range(10))
285
```

Another fun example:

Lets say we want to find first 10 (or any n) pythogorian triplets. A triplet `(x, y, z)` is called pythogorian triplet if `x*x + y*y == z*z`.

It is easy to solve this problem if we know till what value of *z* to test for. But we want to find first n pythogorian triplets.

```
>>> pyt = ((x, y, z) for z in integers() for y in xrange(1, z) for x in range(1,
y) if x*x + y*y == z*z)
>>> take(10, pyt)
[(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15, 17), (12, 16, 20), (15,
20, 25), (7, 24, 25), (10, 24, 26), (20, 21, 29)]
```

5.3.1. Example: Reading multiple files

Lets say we want to write a program that takes a list of filenames as arguments and prints contents of all those files, like `cat` command in unix.

The traditional way to implement it is:

```
def cat(filenames):
    for f in filenames:
        for line in open(f):
            print line,
```

Now, lets say we want to print only the line which has a particular substring, like `grep` command in unix.

```
def grep(pattern, filenames):
    for f in filenames:
        for line in open(f):
            if pattern in line:
                print line,
```

Both these programs have lot of code in common. It is hard to move the common part to a function. But with generators makes it possible to do it.

```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in lines)

def printlines(lines):
    for line in lines:
        print line,

def main(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

The code is much simpler now with each function doing one small thing. We can move all these functions into a separate module and reuse it in other programs.

Problem 2: Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.

Problem 3: Write a function `findfiles` that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.

Problem 4: Write a function to compute the number of python files (.py extension) in a specified directory recursively.

Problem 5: Write a function to compute the total number of lines of code in all python files in the specified directory recursively.

Problem 6: Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

Problem 7: Write a program `split.py`, that takes an integer `n` and a filename as command line arguments and splits the file into multiple small files with each having `n` lines.

5.4. Itertools

The itertools module in the standard library provides lot of intersting tools to work with iterators.

Lets look at some of the interesting functions.

chain – chains multiple iterators together.

```
>>> it1 = iter([1, 2, 3])
>>> it2 = iter([4, 5, 6])
>>> itertools.chain(it1, it2)
[1, 2, 3, 4, 5, 6]
```

izip – iterable version of zip

```
>>> for x, y in itertools.izip(["a", "b", "c"], [1, 2, 3]):
...     print x, y
...
a 1
b 2
c 3
```

Problem 8: Write a function `peep`, that takes an iterator as argument and returns the first element and an equivalant iterator.

```
>>> it = iter(range(5))
>>> x, it1 = peep(it)
>>> print x, list(it1)
0 [0, 1, 2, 3, 4]
```

Problem 9: The built-in function `enumerate` takes an iteratable and returns an iterator over pairs (index, value) for each value in the source.

```
>>> list(enumerate(["a", "b", "c"])
[(0, "a"), (1, "b"), (2, "c")]
>>> for i, c in enumerate(["a", "b", "c"]):
...     print i, c
...
0 a
1 b
2 c
```

Write a function `my_enumerate` that works like `enumerate`.

Problem 10: Implement a function `izip` that works like `itertools.izip`.

Further Reading

- Generator Tricks For System Programers by David Beazly is an excellent in-depth introduction to generators and generator expressions.

Previous Next