Python Iterators

Alan G. Isaac

September 25, 2013

Part I

Iterators and Generators

0.1 Preliminaries

In [1]:
```
%matplotlib inline
import random
import numpy as np
import matplotlib.pyplot as plt
```

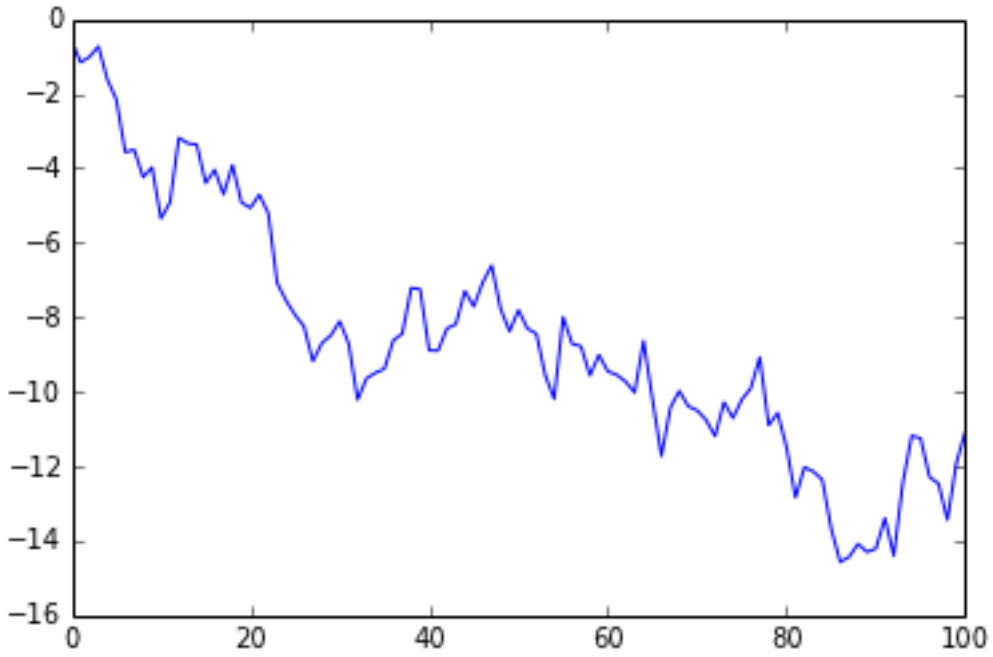
0.2 Iterators

Roughly speaking, an iterator is an object with a `next` method.
But an iterator should also produce an iterator when it `iter` is
applied to it – a requirement we meet by defining an appropriate
`__iter__` method. Our `__iter__` method will
simply return `self`. To illustrate, let us define a simple
random walk iterator.

In [2]:
```
class RandomWalk(object):
    def __init__(self):
        self.val = 0
    def __iter__(self):
        return self
    def next(self):  #Python 2
        self.val += random.normalvariate(0,1)
        return self.val
```

In [4]:
```
rw01 = RandomWalk()
random.seed(314)
data01 = list((next(rw01) for _ in range(101)))

fig, ax = plt.subplots(1,1)
ax.plot(data01)
plt.show()
```



0.3 Generators

Another approach is to use a generator factory. In Python these are called generator functions: functions that return generators. The function definition looks normal, except for the presence of the `yield` keyword.

In [6]:
```
def g_my123():
    yield 1
    yield 2
    yield 3
test = g_my123()
list(test)
```

Out [6]:
```
[1, 2, 3]
```

We can call `next` on a generator to produce its next value. If we do this too many times, we raise a `StopIteration` error.

In [5]:
```
next(test)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent
call last)


<ipython-input-5-911ea584f8be> in <module>()
----> 1 next(test)
```

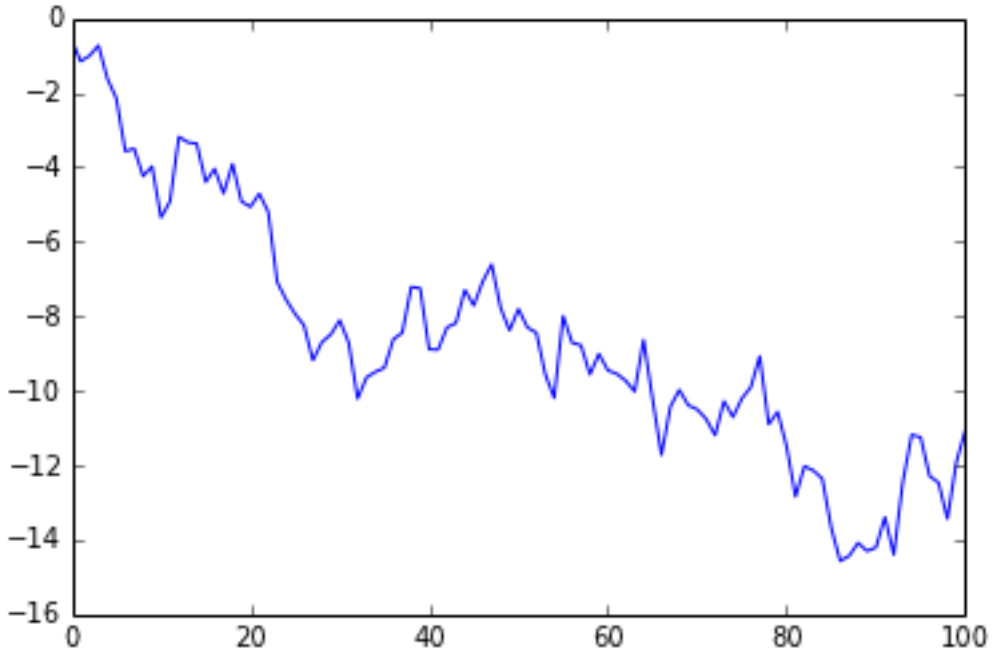
```
StopIteration:
```

Generating a Random Walk

In [5]:
```
def randomwalk():
    val = 0
    while True:
        val += random.normalvariate(0,1)
        yield val
```

In [8]:
```
rw02 = randomwalk()
random.seed(314)
data02 = list(next(rw02) for _ in range(101))

fig, ax = plt.subplots(1,1)
ax.plot(data02)
plt.show()
```



Breaking It Into Pieces

We can break this down into parts. Let us first produce a way to generate a predictable sequence of shocks.

In [9]:
```
def g_shock(maxct=10**3, seed=None):
    prng = random.Random(seed)
    ct = 0
    while (ct < maxct):
        ct += 1
        yield prng.normalvariate(0,1)
list(g_shock(101,314)) == list(g_shock(101,314))
```

```
True
```

Out [9]:

Next we produce cumulative sums for any iterable.

In [10]:
```
def g_cumsum(iterable):
    csum = 0
    for val in iterable:
        csum += val
        yield csum

data01 == list(g_cumsum(g_shock(101, 314)))
```

```
True
```

Out [10]:

For example, we might want to work with the shocks someone else produced, when we try to replicate their work.