Tutorial on Python Iterators and Generators

Norman Matloff
University of California, Davis

April 28, 2007

Contents

1 Iterators

1.1 What Are Iterators? Why Use Them?

Let's start with an example we know from our unit on Python file and directory programming (`http://heather.cs.ucdavis.edu/~matloff/Python/PyFileDir.pdf`). Say we open a file and assign the result to **f**, e.g.

```
f = open('x')
```

Suppose we wish to print out the lengths of the lines of the file.

```
for l in f.readlines():
   print len(l)
```

But the same result can be obtained with

```
for l in f:
   print len(l)
```

The second method has two advantages:

(a) it's simpler and more elegant

(b) a line of the file is not read until it is actually needed

Point (a) becomes even clearer if we take the functional programming approach. The code

```
print map(len,f.readlines())
```

is not as nice as

```
print map(len,f)
```

Point (b) would be of major importance if the file were really large. The first method above would have the entire file in memory, very undesirable. Here we read just one line of the file at a time. Of course, we also could do this by calling **readline()** instead of **readlines()**, but not as simply and elegantly.

In our second method, **f** is serving as an **iterator**. Let's look at the concept more generally.

Recall that a Python **sequence** is roughly like an array in most languages, and takes on two forms—lists and tuples.[1] Sequence operations in Python are much more flexible than in a language like C or C++. One can have a function return a sequence; one can **slice** sequences; one can concatenate sequences; etc.

In this context, an **iterator** looks like a sequence when you use it, but with some major differences:

(a) you usually must write a function which actually constructs that sequence-like object

(b) an element of the "sequence" is not actually produced until you need it

(c) unlike real sequences, an iterator "sequence" can be infinitely long

[1] Recall also that strings are tuples, but with extra properties.

1.2 Example: Fibonacci Numbers

For simplicity, let's start with everyone's favorite computer science example, Fibonacci numbers, as defined by the recursion,

$$f_n = \begin{cases} 1, & \text{if } n = 1, 2 \\ f_{n-1} + f_{n-2}, & \text{if } n > 2 \end{cases} \tag{1}$$

It's easy to write a loop to compute these numbers. But let's try it as an iterator:

```
1  # iterator example; uses Fibonacci numbers, so common in computer
2  # science examples:  f_n = f_{n-1} + f_{n-2}, with f_0 = f_1 = 1
3
4  class fibnum:
5     def __init__(self):
6        self.fn2 = 1  # "f_{n-2}"
7        self.fn1 = 1  # "f_{n-1}"
8     def next(self):  # next() is the heart of any iterator
9        # note the use of the following tuple to not only save lines of
10       # code but also to insure that only the old values of self.fn1 and
11       # self.fn2 are used in assigning the new values
12       (self.fn1,self.fn2,oldfn2) = (self.fn1+self.fn2,self.fn1,self.fn2)
13       return oldfn2
14    def __iter__(self):
15       return self
```

Now here is how we would use the iterator, e.g. to loop with it:

```
1  from fib import *
2
3  def main():
4     f = fibnum()
5     for i in f:
6        print i
7        if i > 20: break
8
9  if __name__ == '__main__':
10    main()
```

By including the method **__iter__()** in our **fibnum** class, we informed the Python interpreter that we wish to use this class as an iterator. We also had to include the method **next()**, which as its name implies, is the mechanism by which the "sequence" is formed. This enabled us to simply place an instance of the class in the **for** loop above. Knowing that **f** is an iterator, the Python interpreter will repeatedly call **f.next()**, assigning the values returned by that function to **i**.

As stated above, the iterator approach often makes for more elegant code. But again, note the importance of not having to compute the entire sequence at once. Having the entire sequence in memory would waste memory and would be impossible in the case of an infinite sequence, as we have here. Our **for** loop above is iterating through an infinite number of iterations—and would do so, if we didn't stop it as we did. But each element of the "sequence" is computed only at the time it is needed.

Rather than being thought of as an "accident," one can use exceptions as an elegant way to end a loop involving an iterator, using the built-in exception type **StopIteration**. For example:

```
1  class fibnum20:
2     def __init__(self):
```

```
3         self.fn2 = 1  # "f_{n-2}"
4         self.fn1 = 1  # "f_{n-1}"
5      def next(self):
6         (self.fn1,self.fn2,oldfn2) = (self.fn1+self.fn2,self.fn1,self.fn2)
7         if oldfn2 > 20: raise StopIteration
8         return oldfn2
9      def __iter__(self):
10        return self
```

```
>>> from fib20 import *
>>> g = fibnum20()
>>> for i in g:
...    i
...
1
1
2
3
5
8
13
```

What happens is that iterating in the loop

```
>>> for i in g:
```

catches the exception **StopIteration**, which makes the looping terminate, and our "sequence" is finite.

You can also make a real sequence out of an iterator's "output" by using the **list()** function, though you of course do have to make sure the iterator produces finite output. For example:

```
>>> from fib20 import *
>>> g = fibnum20()
>>> g
<fib20.fibnum20 instance at 0xb7e6c50c>
>>> list(g)
[1, 1, 2, 3, 5, 8, 13]
```

The functions **sum()**, **max()** and **min()** are built-ins for iterators, e.g.

```
>>> from fib20 import *
>>> g = fibnum20()
>>> sum(g)
33
```

1.3 Example: "Circular" Array

Here's an example of using iterators to make a "circular" array. In our tutorial on Python network programming, `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`, we needed to continually cycle through a list **cs** of client sockets:[2]

```
1  while (1):
2     # get next client, with effect of a circular queue
```

[2] I am slightly modifying it here, by assuming a constant number of clients.

```
3       clnt = cs.pop(0)
4       ...
5       cs.append(clnt)
6       ...
```

Here's how to make an iterator out of it:[3]

```
1   # circular queue
2
3   class cq:  # the argument q is a list
4      def __init__(self,q):
5         self.q = q
6      def __iter__(self):
7         return self
8      def next(self):
9         self.q = self.q[1:] + [self.q[0]]
10        return self.q[-1]
```

Let's test it:

```
>>> from cq import *
>>> x = cq([1,2,3])
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
1
>>> x.next()
2
```

With this, our **while** loop in the network program above would look like this:

```
1   cit = cq(cs)
2   for clnt in cit:
3      # code using clnt
4      ...
```

The code would iterate indefinitely.

Of course, we had to do a bit of work to set this up. But now that we have, we can reuse this code in lots of different applications in the future, and the nice, clear form such as that above,

```
for clnt in cs:
```

adds to ease of programming and readability of code.

1.4 The `itertools` Module

Here you can really treat an infinite iterator like a "sequence," using various tools in this module.

For instance, **iterators.islice()** is handy:

[3] I've also made the code more compact, independent of the change to an iterator.

```
>>> from itertools import *
>>> g = fibnum()
>>> list(islice(g,6))  # slice out the first 6 elements
[1, 1, 2, 3, 5, 8]
```

The general form of **islice()** is

```
itertools.islice(iteratorname, [start], stop, [step])
```

Here we get elements *start*, *start* + **step**, and so on, but ending before element *stop*.

For instance:

```
>>> list(islice(g,3,9,2))
[3, 8, 21]
```

There are also analogs of the **map()** and **filter()** functions which operate on real sequences. The call

```
itertools.imap(f, iter1, iter2, ...)
```

returns the stream **f(iter1[0],iter2[0],...)**, which one can then apply **list()** to.

The call

```
itertools.ifilter(boolean expression, iter)
```

applies the boolean test to each elment of the iterator stream.

2 Generators

2.1 General Structures

Generators are entities which generate iterators! Hence the name.

Speaking very roughly in terms of our goals, a generator is a function that we wish to call repeatedly, but which is unlike an ordinary function in that successive calls to a generator function don't start execution at the beginning of the function. Instead, the current call to a generator function will resume execution right after the spot in the code at which the last call exited, i.e. we "pick up where we left off."

The way this occurs is as follows. One calls the generator itself just once. That returns an iterator. This is a real iterator, with __**iter()**__ and **next()** methods. The latter is essentially the function which implements our "pick up where we left off" goal. We can either call **next()** directly, or use the iterator in a loop.

Note that difference in approach:

- In the case of iterators, a class is recognized by the Python interpreter as an iterator by the presence of the __**iter()**__ and **next()** methods.

- By contrast, with a generator we don't even need to set up a class. We simply write a plain function, with its only distinguishing feature for recognition by the Python interpreter being that we use **yield** instead of **return**.

Note, though, that **yield** and **return** work quite differently from each other. When a **yield** is executed, the Python interpreter records the line number of that statement (there may be several **yield** lines within the same generator). Then, the next time this generator function is called with this same iterator, the function will resume execution at the line following the **yield**.

Here are the key points:

- A **yield** causes an exit from the function, but the next time the function is called, we start "where we left off," i.e. at the line following the **yield** rather than at the beginning of the function.
- All the values of the local variables which existed at the time of the **yield** action are now still intact when we resume.
- There may be several **yield** lines in the same generator.
- We can also have **return** statements, but execution of any such statement will result in a **StopIteration** exception being raised if the **next()** method is called again.
- The **yield** operation returns one argument (or none). That one argument can be a tuple, though. As usual, if there is no ambiguity, you do not have to enclose the tuple in parentheses.

Read the following example carefully, keeping all of the above points in mind:

```
1  # yieldex.py example of yield, return in generator functions
2
3  def gy():
4     x = 2
5     y = 3
6     yield x,y,x+y
7     z = 12
8     yield z/x
9     print z/y
10    return
11
12 def main():
13    g = gy()
14    print g.next()
15    print g.next()
16    print g.next()
17
18 if __name__ == '__main__':
19    main()
```

```
1  % python yieldex.py
2  (2, 3, 5)
3  6
4  4
5  Traceback (most recent call last):
6    File "yieldex.py", line 19, in ?
7      main()
8    File "yieldex.py", line 16, in main
9      print g.next()
10 StopIteration
```

Note that execution of the actual code in the function **gy()**, i.e. the lines

```
x = 2
...
```

does not occur until the first **g.next()** is executed.

2.2 Example: Fibonacci Numbers

As another simple illustration, let's look at the good ol' Fibonacci numbers again:

```
1  # fibg,py, generator example; Fibonacci numbers
2  # f_n = f_{n-1} + f_{n-2}
3
4  def fib():
5     fn2 = 1  # "f_{n-2}"
6     fn1 = 1  # "f_{n-1}"
7     while True:
8        (fn1,fn2,oldfn2) = (fn1+fn2,fn1,fn2)
9        yield oldfn2
```

```
1  >>> from fibg import *
2  >>> g = fib()
3  >>> g.next()
4  1
5  >>> g.next()
6  1
7  >>> g.next()
8  2
9  >>> g.next()
10 3
11 >>> g.next()
12 5
13 >>> g.next()
14 8
```

Note that we do need to resume execution of the function "in the middle," rather than "at the top." We certainly don't want to execute

```
fn2 = 1
```

again, for instance. Indeed, a key point is that the local variables **fn1** and **fn2** retain their values between calls. This is what allowed us to get away with using just a function instead of a class. This is simpler and cleaner than the class-based approach. For instance, in the code here we refer to **fn1** instead of **self.fn1** as we did in our class-based version in Section 1.2. In more complicated functions, all these simplifications would add up to a major improvement in readability.

This property of retaining locals between calls is like that of locals declared as **static** in C.[4]). Note, though, that in Python we might set up several instances of a given generator, each instance maintaining different values for the locals. To do this in C, we need to have arrays of the locals, indexed by the instance number.

[4]If you need review of this in the C context, make sure to check a C book, or the "C portion" of a C++ book. It's a very important concept

2.3 Example: Word Fetcher

The following is a producer/consumer example. The producer, **getword()**, gets words from a text file, feeding them one at a time to the consumer.[5] In the test here, the consumer is **testgw.py**.

```
# getword.py

# the function getword() reads from the text file fl, returning one word
# at a time; will not return a word until an entire line has been read

def getword(fl):
   for line in fl:
      for word in line.split():
         yield word
   return
```

```
# test of getword; counts words and computes average length of words
# usage:  python testgw.py [filename]
# (stdin) is assumed if no file is specified)

from getword import *

def main():
   import sys
   # determine which file we'll evaluate
   try:
      f = open(sys.argv[1])
   except:
      f = sys.stdin
   # generate the iterator
   w = getword(f)
   wcount = 0
   wltot = 0
   for wrd in w:
      wcount += 1
      wltot += len(wrd)
   print "%d words, average length %f" % (wcount,wltot/float(wcount))

if __name__ == '__main__':
   main()
```

2.4 Mutiple Iterators from the Same Generator

Note our phrasing earlier (emphasis added):

> ...the next time this generator function is called *with this same iterator*, the function will resume execution at the line following the **yield**

Suppose for instance that we have two sorted text files, one word per line, and we wish to merge them into a combined sorted file. We could use our **getword()** function above, setting up two iterators, one for each file. Note that we might reach the end of one file before the other. We would then continue with the other file by itself. To deal with this, we would have to test for the **StopIteration** exception to sense when we've come to the end of a file.

[5] I thank C. Osterwisch for this much improved version of the code I had here originally.

2.5 Modularity/Reusability

You may have noticed in the last example a similarity to Unix pipes. At the shell level, we can do a lot of everyday tasks by simply chaining together several shell commands into a pipe. For instance, say I want to find out how many lines in the file **g** contain the word 'Davis'. I could do this:

```
% grep Davis x | wc -l
```

The command **wc** ("word count"), with its **-l** option, counts lines. So, **grep** would find the file's lines that I want, and pass them on to **wc** via the pipe, after which **wc** would count them.

Similarly, the **getword()** function above is producing output which then is used as input by our program **testgw.py**. We could chain several generators together in a "pipe."

This shows that one of the biggest advantages of using iterators, and especially generators, is modularity and reusability. Once you write a few tools like this, you can keep making use of them in lots of applications that you write. Of course, we could still do that with ordinary functions, but without the compact elegance and clarity that we get from iterators, e.g.

```
for w in word:
```

in the example above.

2.6 Coroutines

The term **coroutines** in computer science refers to subroutines that alternate in execution. Subroutine A will run for a while, then subroutine B will run for a while, then A again, and so on. Each a subroutine runs, it will resume execution right where it left off before—just like Python generators.

Basically coroutines are threads, but of the nonpreemptive type. In other words, a coroutine will continue executing until it voluntarily relinquishes the CPU. (Of course, this doesn't count timesharing. We are only discussing flow of control among the threads of one program.) In "ordinary" threads, the timing of the passing of control from one thread to another is to various degrees random.

The major advantage of using nonpreemptive threads is that you do not need locks. This makes your code a lot simpler and cleaner, and much easier to debug. (The randomness alone makes ordinary threads really tough to debug.)

In this section, I will show you two examples of Python coroutines. The first is a library class I wrote, **thrd**, which serves as a Python nonpreemptive threads library. The second example is SimPy, a well-known Python discrete-event simulation library written by Klaus Muller and Tony Vignaux.

2.6.1 My thrd Class

Though most threading systems are preemptive, there are some prominent exceptions. The GNU PTH library, for instance, is nonpreemptive and supports C/C++. Another example is the threads library in the Ruby scripting language.

Generators make it easy to develop a nonpreemptive threads package in Python. The **yield** construct is a natural way to relinquish the CPU, and one writes the threads manager to give a thread a turn by simply calling **i.next()**, where **i** is the iterator for the thread. That's what I've done here.

As an example of use of **thrd**, we'll take the "string build" example presented in our units on networks and threading, available at `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf` and `http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf`. Clients send characters one at a time to a server, which accumulates them in a string, which it echoes back to the clients.

There are two major issues in the example. First, we must deal with the fact that we have asynchronous I/O; the server doesn't know which client it will hear from next. Second, we must make sure that the accumulated string is always updated atomically.

Here we will use nonblocking I/O to address the issue of asynchroneity. But atomicity will be no problem at all. Again, since threads are never interrupted, we do not need locks. Here is the code for the server:

```
1  # simple illustration of thrd module
2
3  # multiple clients connect to server; each client repeatedly sends a
4  # letter k, which the server adds to a global string v and echos back
5  # to the client; k = '' means the client is dropping out; when all
6  # clients are gone, server prints final value of v
7
8  # this is the server
9
10 import socket
11 import sys
12 from pth import *
13
14 class glbs:  # globals
15    v = ''  # the string we are building up from the clients
16
17 class serveclient(thrd):
18    def __init__(self,id,c):
19       thrd.__init__(self,id)
20       self.c = c[0]  # socket for this client
21       self.c.send('c')  # confirm connection
22    def run(self):
23       while 1:
24          # receive letter or EOF signal from c
25          try:
26             k = self.c.recv(1)
27             if k == '': break
28             # concatenate v with k, but do NOT need a lock
29             glbs.v += k
30             self.c.send(glbs.v)
31          except:
32             pass
33          yield 'clnt loop', 'pause'
34       self.c.close()
35
36 def main():
37    lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
38    port = int(sys.argv[1])  # server port number
39    lstn.bind(('', port))
40    lstn.listen(5)
41    # initialize concatenated string, v
42    glbs.v = ''
43    # number of clients
44    nclnt = 2
45    # accept calls from the clients
46    for i in range(nclnt):
47       (clnt,ap) = lstn.accept()
```

```
48          clnt.setblocking(0)  # set client socket to be nonblocking
49          # start thread for this client, with the first argument being a
50          # string ID I choose for this thread, and the second argument begin
51          # (a tuple consisting of) the socket
52          t = serveclient('client '+str(i),(clnt,))
53       # shut down the server socket, since it's not needed anymore
54       lstn.close()
55       # start the threads; the call will block until all threads are done
56       thrd.tmgr()
57       print 'the final value of v is', glbs.v
58
59    if __name__ == '__main__':  main()
```

Here is the client (which of course is not threaded):

```
1   # simple illustration of thrd module
2
3   # two clients connect to server; each client repeatedly sends a letter,
4   # stored in the variable k, which the server appends to a global string
5   # v, and reports v to the client; k = '' means the client is dropping
6   # out; when all clients are gone, server prints the final string v
7
8   # this is the client; usage is
9
10  #    python clnt.py server_address port_number
11
12  import socket
13  import sys
14
15  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
16  host = sys.argv[1]  # server address
17  port = int(sys.argv[2])  # server port
18  s.connect((host, port))
19
20  confirm = s.recv(1)
21  print confirm
22
23  while(1):
24     # get letter
25     k = raw_input('enter a letter:')
26     s.send(k)  # send k to server
27     # if stop signal, then leave loop
28     if k == '': break
29     v = s.recv(1024)  # receive v from server (up to 1024 bytes)
30     print v
31
32  s.close() # close socket
```

Note that as with the Python **threading** module, the user must write a function named **run** which will override the one built in to the **thrd** class. As before, that function describes the action of the thread. The difference here, though, is that now this function is a generator, as you (and the Python interpreter) can tell from the presence of the **yield** statement.

There is a separate thread for each client. The thread for a given client will repeatedly execute the following cycle:

- Try to read a character from the client.
- Process the character if there is one.
- Yield, allowing the thread for another client to run.

Again, since a thread will run until it hits a **yield**, we don't need locks.

Just as is the case with Python's ordinary threads, **thrd** is good mainly for I/O-bound applications. While one I/O action is being done in one thread, we can start another one in another thread. A common example would be a Web server. But those applications would be too huge to deal with in this tutorial, so we have that very simple toy example above.

Below is another toy example, even more contrived, but again presented here because it is simple, and because it illustrates the set/wait constructs not included in the last example. There is really no way to describe the actions it takes, except to say that it is designed to exercise most of the possible **thrd** operations. Just look at the output shown below, and then see how the code works to produce that output.

```
a1 starts
a1 x: 6
a1 pauses
a2 starts
a2 x: 7
a2 pauses
b starts
b pauses
c1 starts
c1 waits for a1-ev
c2 starts
c2 waits for a1-ev
a1 z: 19
a1 waits for b-ev
a2 z: 21
a2 waits for b-ev
b.v: 8
b sets b-ev
a1 z: 19
a1 sets a1-ev for all
c1 quits
events:
b-ev: a2
a1-ev:
c2 quits
events:
b-ev: a2
a1-ev:
b sets b-ev but stays
b quits
a2 z: 21
a2 quits
a1 quits
```

Here is the code:

```
from pth import *

class a(thrd):
   def __init__(self,thrid):
      thrd.__init__(self,thrid)
      self.x = None
      self.y = None
      self.z = None
      self.num = int(self.id[1])
   def run(self):
      print self.id, 'starts'
      self.x = 5+self.num
      self.y = 12+self.num
      print self.id, 'x:', self.x
```

```
      print self.id, 'pauses'
      yield '1','pause'
      self.z = self.x + self.y
      print self.id, 'z:', self.z
      print self.id, 'waits for b-ev'
      yield '2','wait','b-ev'
      print self.id, 'z:', self.z
      if self.id == 'a1':
         print 'a1 sets a1-ev for all'
         yield '2a','set_all','a1-ev'
      print self.id, 'quits'
      yield '3','quit'

class b(thrd):
   def __init__(self,thrid):
      thrd.__init__(self,thrid)
      self.u = None
      self.v = None
   def run(self):
      print 'b starts'
      self.u = 5
      print 'b pauses'
      yield '11','pause'
      self.v = 8
      print 'b.v:', self.v
      print 'b sets b-ev'
      yield '12','set','b-ev'
      print 'b sets b-ev but stays'
      yield 'uv','set_but_stay','b-ev'
      print 'b quits'
      yield 'our last one','quit'

class c(thrd):
   def __init__(self,thrid):
      thrd.__init__(self,thrid)
   def run(self):
      print self.id, 'starts'
      print self.id, 'waits for a1-ev'
      yield 'cwait','wait','a1-ev'
      print self.id, 'quits'
      thrd.prevs()
      yield 'cquit','quit'

def main():
    ta1 = a('a1')
    ta2 = a('a2')
    tb = b('b')
    tc1 = c('c1')
    tc2 = c('c2')
    thrd.tmgr()

if __name__ == '__main__': main()
```

Now, how is all this done. Below is the code for the **thrd** library.

First read the comments at the top of the file, and then the **__init()__** code. Then glance at the code for the threads manager . The latter repeatedly does the following:

- get the first thread in the runnable list, **thr**
- have it run until it hits a **yield**, by calling **thr.itr.next()**
- take whatever action (pause, wait, set, etc.) that the thread requested when it yielded

Then you should be able to follow the **thrd** member functions fairly easily.

```
1  # pth.py:  non-preemptive threads for Python; inspired by the GNU PTH
2  # package for C/C++
3
4  # typical application will have one class for each type of thread; its
5  # main() will set up the threads as instances of the classes, and lastly
6  # will call thrd.tmgr()
7
8  # each thread type is a subclass of the class thrd; in that subclass,
9  # the user must override thrd.run(), with the code consisting of the
10 # actions the thread will take
11
12 # threads actions are triggered by the Python yield construct, in the
13 # following format:
14
15 #    yield yieldID, action_string [, arguments]
16
17 # the yieldID is for application code debugging purposes
18
19 # possible actions:
20
21 #    yield yieldID, 'pause':
22 #       thread relinquishes this turn, rejoins runnable list at the end
23
24 #    yield yieldID, 'wait', eventID:
25 #       thread changes state to 'waiting', joins end of queue for
26 #       the given event
27
28 #    yield yieldID, 'set', eventID:
29 #       thread sets the given event, rejoins runnable list at the end;
30 #       the thread, if any, at head of queue for this event is inserted
31 #       at the head of the runnable list
32
33 #    yield yieldID, 'set_but_stay', eventID:
34 #       thread sets the given event, but remains at head of runnable list;
35 #       thread, if any, at head of queue for the event is inserted in
36 #       runnable list following the head
37
38 #    yield yieldID, 'set_all', eventID:
39 #       thread sets the given event, rejoins runnable list at the end;
40 #       all threads in queue for the event are inserted at the head of
41 #       the runnable list, in the same order they had in the queue
42
43 #    yield yieldID, 'quit':
44 #       thread exits
45
46 class thrd:
47
48    runlst = []  # runnable thread list
49    evnts = {}  # a key is an event ID, a string; value is a list of
50                # threads waiting for that event
51    waitlst = []  # waiting thread list
52    didyield = None  # thread that last performed a yield op; for
53                     # application code debugging purposes
54
55    def __init__(self,id):
56       self.id = id  # user-supplied string
57       self.state = 'runnable'  # the other possible state is 'waiting'
58       self.yieldact = ''  # action at last yield; for application code
59                           # debugging purposes
60       self.waitevnt = ''  # what event this thread is waiting for, if any;
61                           # for application code debugging purposes
62       self.itr = self.run()  # this thread's iterator
63       thrd.runlst.append(self)
64
65    def run(self):  # stub, must override
```

```
66          pass
67
68      # triggered by:  yield yieldID, 'pause'
69      def do_pause(self,yv):
70          del thrd.runlst[0]
71          thrd.runlst.append(self)
72
73      # triggered by:  yield yieldID, 'wait', eventID
74      def do_wait(self,yv):
75          del thrd.runlst[0]
76          self.state = 'waiting'
77          self.waitevnt = yv[2]
78          # check to see if this is a new event
79          if yv[2] not in thrd.evnts.keys():
80              thrd.evnts[yv[2]] = [self]
81          else:
82              thrd.evnts[yv[2]].append(self)
83          thrd.waitlst.append(self)
84
85      # reactivate first thread waiting for this event, and place it at
86      # position pos of runlst
87      def react(ev,pos):
88          thr = thrd.evnts[ev].pop(0)
89          thr.state = 'runnable'
90          thr.waitevnt = ''
91          thrd.waitlst.remove(thr)
92          thrd.runlst.insert(pos,thr)
93      react = staticmethod(react)
94
95      # triggered by:  yield yieldID, 'set', eventID
96      def do_set(thr,yv):
97          del thrd.runlst[0]
98          thrd.runlst.append(thr)
99          thrd.react(yv[2],0)
100     do_set = staticmethod(do_set)
101
102     # triggered by:  yield yieldID, 'set_but_stay'
103     def do_set_but_stay(thr,yv):
104         thrd.react(yv[2],1)
105     do_set_but_stay = staticmethod(do_set_but_stay)
106
107     # triggered by:  yield yieldID, 'set_all', eventID
108     def do_set_all(self,yv):
109         del thrd.runlst[0]
110         ev = yv[2]
111         for i in range(len(thrd.evnts[ev])):
112             thrd.react(ev,i)
113         thrd.runlst.append(self)
114
115     # triggered by:  yield yieldID, 'quit'
116     def do_quit(self,yv):
117         del thrd.runlst[0]
118
119     # for application code debugging
120     # prints info about a thread
121     def prthr(self):
122         print 'ID: %s, state: %s, ev: %s, yldact: %s' % \
123             (self.id, self.state, self.waitevnt, self.yieldact)
124
125     # for application code debugging
126     # print info on all threads
127     def prthrs():
128         print 'runlst:'
129         for t in thrd.runlst:
130             t.prthr()
131         print 'waiting list:'
132         for t in thrd.waitlst:
133             thrd.prthr(t)
```

```
134     prthrs = staticmethod(prthrs)
135
136     # for application code debugging
137     # printf info on all events
138     def prevs():
139        print 'events:'
140        for eid in thrd.evnts.keys():
141           print '%s:' % eid,
142           for thr in thrd.evnts[eid]:
143              print thr.id,
144           print
145     prevs = staticmethod(prevs)
146
147     # threads manager
148     def tmgr():
149        # while still have runnable threads, cycle repeatedly through them
150        while (thrd.runlst):
151           # get next thread
152           thr = thrd.runlst[0]
153           # call it
154           yieldvalue = thr.itr.next()
155           # the above call to next() runs the thread until a yield, with
156           # the latter returning yieldvalue
157           thr.yieldID = yieldvalue[0]
158           thrd.didyield = thr
159           # call the function requested in the yield
160           yv1 = yieldvalue[1]  # requested action
161           thr.yieldact = yv1
162           actftn = eval('thrd.do_'+yv1)
163           actftn(thr,yieldvalue)
164     tmgr = staticmethod(tmgr)
```

2.6.2 The SimPy Discrete Event Simulation Library

In discrete event simulation (DES), we are modeling discontinuous changes in the system state. We may be simulating a queuing system, for example, and since the number of jobs in the queue is an integer, the number will be incremented by an integer value, typically 1 or -1.[6] By contrast, if we are modeling a weather system, variables such as temperature change continuously.

SimPy is a widely used open-source Python library for DES. Following is an example of its use:

```
1  #!/usr/bin/env python
2
3  # MachRep.py
4
5  # SimPy example:  Two machines, but sometimes break down.  Up time is
6  # exponentially distributed with mean 1.0, and repair time is
7  # exponentially distributed with mean 0.5.  In this example, there is
8  # only one repairperson, so the two machines cannot be repaired
9  # simultaneously if they are down at the same time.
10
11 # In addition to finding the long-run proportion of up time, let's also
12 # find the long-run proportion of the time that a given machine does not
13 # have immediate access to the repairperson when the machine breaks
14 # down.  Output values should be about 0.6 and 0.67.
15
16 from SimPy.Simulation import *
17 from random import Random,expovariate,uniform
18
19 class G:  # globals
20    Rnd = Random(12345)
```

[6] **Batch queues** may take several jobs at a time, but the increment is still integer-valued.

```
21      # create the repairperson
22      RepairPerson = Resource(1)
23
24  class MachineClass(Process):
25      TotalUpTime = 0.0  # total up time for all machines
26      NRep = 0  # number of times the machines have broken down
27      NImmedRep = 0  # number of breakdowns in which the machine
28                     # started repair service right away
29      UpRate = 1/1.0  # breakdown rate
30      RepairRate = 1/0.5  # repair rate
31      # the following two variables are not actually used, but are useful
32      # for debugging purposes
33      NextID = 0  # next available ID number for MachineClass objects
34      NUp = 0  # number of machines currently up
35      def __init__(self):
36          Process.__init__(self)
37          self.StartUpTime = 0.0  # time the current up period stated
38          self.ID = MachineClass.NextID   # ID for this MachineClass object
39          MachineClass.NextID += 1
40          MachineClass.NUp += 1  # machines start in the up mode
41      def Run(self):
42          while 1:
43              self.StartUpTime = now()
44              yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
45              MachineClass.TotalUpTime += now() - self.StartUpTime
46              # update number of breakdowns
47              MachineClass.NRep += 1
48              # check whether we get repair service immediately
49              if G.RepairPerson.n == 1:
50                  MachineClass.NImmedRep += 1
51              # need to request, and possibly queue for, the repairperson
52              yield request,self,G.RepairPerson
53              # OK, we've obtained access to the repairperson; now
54              # hold for repair time
55              yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
56              # release the repairperson
57              yield release,self,G.RepairPerson
58
59  def main():
60      initialize()
61      # set up the two machine processes
62      for I in range(2):
63          M = MachineClass()
64          activate(M,M.Run())
65      MaxSimtime = 10000.0
66      simulate(until=MaxSimtime)
67      print 'proportion of up time:', MachineClass.TotalUpTime/(2*MaxSimtime)
68      print 'proportion of times repair was immediate:', \
69          float(MachineClass.NImmedRep)/MachineClass.NRep
70
71  if __name__ == '__main__':  main()
```

There is a lot here, but basically it is similar to the **thrd** class we saw above. If you were to look at the SimPy internal code, **SimPy.Simulation.py**, you would see that a large amount of it looks like the code in **thrd**. In fact, the SimPy library could be rewritten on top of **thrd**, greatly reducing the size of the library. That would make future changes to the library easier, and would even make it easier to convert SimPy to some other language, say Ruby.

Read the comments in the first few lines of the code to see what kind of system this program is modeling before going further.

Now, let's see the details.

SimPy's thread class is **Process**. The application programmer writes one or more subclasses of this one to

serve as thread classes. Similar to the case for the **thrd** and **threading** classes, the subclasses of **Process** must include a method **Run()**, which describes the actions of the thread. The SimPy method **activate()** is used to add a thread to the run list.

The main new ingredient here is the notion of simulated time. The current simulated time is stored in the variable **Simulation._t**. Each time an event is created, via execution of a statement like

```
yield hold, self, holdtime
```

SimPy schedules the event to occur **holdtime** time units from now, i.e. at time **_t+holdtime**. What I mean by "schedule" here is that SimPy maintains an internal data structure which stores all future events, ordered by their occurrence times. Let's call this the scheduled events structure, SES. Note that the elements in SES are threads, i.e. instances of the class **Process**. A new event will be inserted into the SES at the proper place in terms of time ordering.

The main loop in SimPy repeatedly cycles through the following:

- Remove the earliest event, say **v**, from SES.
- Advance the simulated time clock **Simulation._t** to the occurrence time of **v**.
- Call the iterator for **v**, i.e. the iterator for the **Run()** generator of that thread.
- After **Run()** does a **yield**, act on whatever operation it requests, such as **hold**.

In simulation programming, we often need to have one entity wait for some event to occur. In our example here, if one machine goes down while the other is being repaired, the newly-broken machine will need to wait for the repairperson to become available. Clearly this is like the condition variables construct in most threads packages, including the **wait** and **set** operations in **thrd**, albeit at a somewhat higher level.

Specifically, SimPy includes a **Resource** class. In our case here, the resource is the repairperson. When a line like

```
yield request,self,G.RepairPerson
```

is executed, SimPy will look at the internal data structure in which SimPy stores the queue for the repairperson. If it is empty, the thread that made the request will acquire access to the repairperson, and will be kept on the runnable list. If there are threads in the queue (here, there would be at most one), then the thread which made the request will be removed from the runnable list, and added to the queue. Later, when a statement like

```
yield release,self,G.RepairPerson
```

is executed by the thread currently accessing the repairperson, SimPy will check its queue, and if the queue is nonempty, SimPy will remove the first thread from the queue, and add it back to the runnable list.

Since the simulated time variable **Simulation._t** is in a separate module, we cannot access it directly. Thus SimPy includes a "getter" function, **now()**, which returns the value of **Simulation._t**.

Most discrete event simulation applications are stochastic in nature, such as we see here with the random up and repair times for the machines. Thus most SimPy programs import the Python **random** module, as in this example.