Introduction to SimPy Internals

Norm Matloff

February 20, 2008

Contents

1 Purpose

In simulation (and other) languages, one often wonders "What does this operation REALLY do?" The description in the documentation may not be fully clear, say concerning the behavior of the operation in certain specialized situations. But in the case of open source software like SimPy, we can actually go into the code to see what the operation really does.

Another reason why access to the language's internals is often useful is that it can aid our debugging activities. We can check the values of the internal data structures, and so on.

Accordingly, this unit will be devoted to introducing the basics of SimPy internals. We will use SimPy version 1.9 as our example.

2 Python Generators

SimPy is built around Python **generators**, which are special kinds of Python functions. Following will be a quick overview of generators, sufficient for our purposes here. If you wish to learn more about generators, see the generators unit in my Python tutorial, at my Python tutorials Web site, `http://heather.cs.ucdavis.edu/˜matloff/python.html`.

Speaking very roughly in terms of usage, 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."

Here is a concrete example:

```
# yieldex.py example of yield, return in generator functions

def gy():
   x = 2
   y = 3
   yield x,y,x+y
   z = 12
   yield z/x
   print z/y
   return

def main():
   g = gy()
   print g.next()  # prints x, y and x+y
   print g.next()  # prints z/x
   print g.next()  # causes the exception

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

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

Here is what happened in the execution of that program:

- As with any Python program, the Python interpreter started execution at the top of the file. When the interpreter sees free-standing code, it executes that code, but if it encounters a function definition, it records it. In particular, the intepreter notices that the function **gy()** contains a **yield** statement, and thus records that this function is a generator rather than an ordinary function. Note carefully that the function has NOT been executed yet at this point.

- The line

  ```
  g = gy()
  ```

  creates a Python **iterator**, assigning it to **g**. Again, to learn the details on iterators, you can read my tutorial above, but all you need to know is that **g** is a certain kind of object which includes a member function named **next()**, and that this function will be our vehicle through which to call **gy()**. Note carefully that **gy()** STILL has not been executed yet at this point.

- The three statements

  ```
  print g.next()
  print g.next()
  print g.next()
  ```

  call **gy()** three times, in each case printing out the value returned by that function, either through **yield** or the traditional **return**.

- With the first call, only the lines

  ```
  x = 2
  y = 3
  yield x,y,x+y
  ```

  are executed. The **yield** acts somewhat like a classical return, in the sense that (a) control passes back to the caller, in this case **main()**, and (b) a value is returned, in this case the tuple **(x,y,x+y)**.[1] This results in (2,3,5) being printed out.

  But the difference between **yield** and **return** is that **yield** also records the point at which we left the generator. In this case here, it means that it will be recorded that our **yield** operation was executed at the first of the two **yield** statements in **gy()**.

- The second call to **g.next()** in **main()** will therefore begin right after the last **yield**, meaning that this second call will begin at the line

  ```
  z = 12
  ```

  instead of at the line

  ```
  x = 2
  ```

  Moreover, the values of the local variables, here **x** and **y**,[2] will be retained; for instance, **y** will still be 3.

- Execution will then proceed through the next **yield**,

  ```
  yield z/x
  ```

  This again will return control to the caller, **main()**, along with the return value **z/x**. Again, it will be noted that the **yield** which executed this time was the second **yield**.

- The third call to **g.next()** causes an execution error. It is treated as an error because a call to a **next()** function for a generator assumes that another **yield** will be encountered, which wasn't the case here. We could have our code sense for this **StopIteration** condition by using Python's **try** construct.

[1] Recall that the parentheses in a tuple are optional if no ambiguity would result from omitting them.

[2] The local **z** has not come into existence yet.

3 How SimPy Works

Armed with our knowledge of generators, we can now take a look inside of SimPy. I've included the source code, consisting of the file **Simulation.py** for version 1.9 of SimPy, in an appendix to this document.

3.1 Running Example

Here and below, let's suppose we have a class in our application code named **X**, which is a subclass of **Process**, and whose PEM is named **Run()**, and that we have created an instance of **X** named **XInst**.

The key point to note is that since **Run()** contains one or more **yield** statements, the Python interpreter recognizes it as a generator. Thus the call **XInst.Run()** within our call to **activate()** (see below) returns an iterator. I'll refer to this iterator here as **XIt** for convenience, though you'll see presently that the SimPy code refers to it in another way. But the point is that **XIt** will be our thread.

3.2 How `initialize()` Works

This function does surprisingly little. Its main actions are to set the global variables **_t**, **_e** and **_stop**, which play the following roles:

- The global **_t** stores the simulated time, initialized to 0. (The application API **now()** simply returns **_t**.)
- The global **_e** is an instance of the class **__Evlist**. One of the member variables of that class is **events**, which is the event list.
- The global **_stop** is a flag to stop the simulation. For example, it is set when **stopSimulation()** is called.

3.3 How `activate()` Works

What happens when our application code executes the following line?

```
activate(XInst,XInst.Run())
```

The definition of **activate()** begins with

```
def activate(obj,process,at="undefined",delay="undefined",prior=False):
```

so in our call

```
activate(XInst,XInst.Run())
```

the formal parameter **obj** will be **XInst**, an instance of a subclass of **Process**, and **process** will be our iterator **XIt**. (As you can see, we have not used the optional named parameters here.)

At this point **activate()** executes its code

```
obj._nextpoint=process
```

Recall that our class **X** is a subclass of SimPy's **Process**. One of the member variables of the latter is **_nextpoint**, and you can now see that it will be our iterator, i.e. our thread. The name of this member variable alludes to the fact that each successive call to a generator "picks up where we last left off." The variable's name can thus be thought of as an abbreviation for "point at which to execute next."

Finally, **activate()** sets **zeit** to the current simulated time **_t**. (The more general usage of **activate()** allows starting a thread later than the current time, but let's keep things simple here.)

Then **activate()** executes

```
_e._post(obj,at=zeit,prior=prior)
```

Here is what that does: Recall that **_e** is the object of class **__Evlist**, which contains our event list. A member function in that class is **_post()**, whose role is to add ("post") an event to the event list. In our case here, there is no real event, but the code will add an artificial event for this thread. The time for this artificial event will be the current time. The effect of this will be that the first execution of this thread will occur "immediately," meaning at the current simulated time. This is what gets the ball rolling for this thread.

3.4 How `simulate()` Works

3.4.1 The Core `while` Loop

The core of **simulate()** consists of a **while** loop which begins with

```
while not _stop and _t<=_endtime:
```

Here **_endtime** is the maximum simulated time set by the application code, and you'll recall that **_stop** is a flag that tells SimPy to stop the simulation.

In each iteration of this **while** loop, the code pulls the event with the earliest simulated time from the event list, updates the current simulated time to that time, and then calls the iterator associated with that event. Remember, that iterator is our thread, so calling it will cause the thread to resume execution. You will see more details in the next section.

3.4.2 Call to `_nextev()`

A key statement near the top of the core **while** loop of **simulate()** is

```
a=nextev()
```

Here **nextev** is an alternate name the authors of SimPy gave to a member function of the **__Evlist** class, **_nextev()**.

The function **_nextev()** extracts the next event, acts on it (e.g. updating the simulated clock), and then has the event's associated thread resume execution until it next hits a **yield**. The latter causes a return to the caller. That returned value consists of a tuple that in the case of our example class **X** above will be **(yield_value,XInst)**, where **yield_value** is the tuple returned by the thread. Following are some of the details.

This version of SimPy stores the events in a heap, using the Python library **heapq**. The latter stores a heap in a Python list, which in our case here is the member variable **timestamps** in the **__Evlist** class. Here is the line within **_nextev()** that extracts the earliest event:

```
(_tnotice, p,nextEvent,cancelled) = hq.heappop(self.timestamps)
```

That variable **_tnotice** now contains the time for this event. The function then updates the simulated time to that time, and checks to see whether the simulation's specified duration has been reached:

```
      _t = _tnotice
      if _t > _endtime:
           _t = _endtime
           _stop = True
```

Eventually this function **_nextev()** executes the statement

```
resultTuple = nextEvent._nextpoint.next()
```

Again, recall that **_nextpoint** is the iterator for this thread. Thus this statement will call the iterator, which causes the thread to resume execution. As noted above, the thread will eventually encounter a **yield**, returning execution to the above statement, and assigning to **resultTuple** the value returned by the **yield**.

Let's recall what **resultTuple** will look like. For instance the statement

```
yield hold,self,0.6
```

returns the 3-tuple **(hold,self,0.6)**, where **hold** is a numerical code, from a set defined in **Simulation.py**:

```
# yield keywords
hold=1
passivate=2
request=3
release=4
waitevent=5
queueevent=6
waituntil=7
get=8
put=9
```

Finally **_nextev()** executes

```
return (resultTuple, nextEvent)
```

where as mentioned, **nextEvent** is our **Process** instance, e.g. **XInst** in our example above. Note that at this point, we have started to set up the next event for this thread, in the information contained in that return tuple. Now we must add it to the event list.

3.4.3 How a New Event Gets Added to the Event List

After calling and performing some checks, **_nextev()**, **simulate()** then executes

```
command = a[0][0]
dispatch[command](a)
```

Here's how what happens: Recall that **a** is the object returned by our call to **_nextev()** that we extracted from the event list, and that *inter alia* it contains the tuple returned when this thread last executed a **yield**. The first element of that tuple will be one of **hold**, **request** etc. This is the basis for formulating our next event, as follows.

SimPy defines a Python dictionary **dispatch** of functions, which serves as a lookup table:

```
dispatch={hold:holdfunc,request:requestfunc,release:releasefunc, \
   passivate:passivatefunc,waitevent:waitevfunc,queueevent:queueevfunc, \
   waituntil:waituntilfunc,get:getfunc,put:putfunc}
```

So, the code

```
command = a[0][0]
dispatch[command](a)
```

has the effect of calling **holdfunc** in the case of **yield hold**, **requestfunc** in the case of **yield request** and so on. Those functions in turn calls others that do the real work. For instance, **holdfunc()** in turn calls **_hold()**, which does

```
_e._post(what=who,at=_t+delay)
```

As you recall, the function **_post()** adds a new event to the event list. The argument **who** here is our event, say **XInst**, and **delay** is the time that **XInst.Run()** asked to hold in its **yield hold** statement, say 0.6. So, you can see that the code above is scheduling an event 0.6 amount of time from now, which is exactly what we want. **XInst**'s **nextTime** field (inherited from the **Process** class) will then be set to **_t+delay**

The function **_post()** adds this new event to the event list, via its line

```
hq.heappush(self.timestamps,what._rec)
```

As mentioned, the heap **_e.timestamps** is a Python list, consisting of instances of **Process** subclases, i.e. consisting of threads. So, we're adding our new event, **what._rec**, to the events heap.

3.5 How `Resource()`, `yield request` and `yield release` Work

Suppose our application code also sets up some resources:

```
R = Resource(2)
```

Recall that **Resource** is a SimPy class, so here we are calling that class' constructor with an argument of 2, meaning that we want two servers or machines or whatever. The constructor includes code

```
self.capacity=capacity  # resource units in this resource
...
self.n=capacity         # uncommitted resource units
```

The formal parameter **capacity** has the actual value 2 in our example here, and as you can see, it is now stored in a member variable of **Process** of the same name. Furthermore, the member variable **n**, which stores the current number of free units of the resource, is initially set to the capacity, i.e. all units are assumed available at the outset.

At this time, the constructor also sets up two other member variables (and more we aren't covering here):

- **waitQ**, the queue of jobs waiting to a unit of this resource
- **activeQ**, the list of jobs currently using a unit of this resource

For **yield request**, **simulate()** calls the function **_request()**. The key code there is, for the non-preemption case,

```
if self.n == 0:
    self.waitQ.enter(obj)
    # passivate queuing process
    obj._nextTime=None
else:
    self.n -= 1
    self.activeQ.enter(obj)
    _e._post(obj,at=_t,prior=1)
```

As you can see, if there are no available units, we add the thread to the queue for this resource, and passivate the thread. But if there is an available unit, the code creates an artificial event, to be executed immediately (as with **activate()**, this is immediate in the sense of being at the same simulated time), and adds it to the event list.

Note that the way that passivation is done is to simply set the thread's **nextTime** field (time of the next event for this thread) to None. This is the way **yield passivate** is handled too:

```
def _passivate(self,a):
     a[0][1]._nextTime=None
```

On the other hand, if there are units available, we grab one, thus decrementing **n** by 1, add the thread to the list of threads currently using the units, and then add this thread to the event list. Since its event time will be **now()**, it will start right back up again immediately in the sense of simulated time, though it may not be the next thread to run.

When a **yield release** statement is executed by the application code, the natural actions are then taken by the function **_release()**:

```
       self.n += 1
       self.activeQ.remove(arg[1])
```

```
        #reactivate first waiting requestor if any; assign Resource to it
        if self.waitQ:
             obj=self.waitQ.leave()
             self.n -= 1              #assign 1 resource unit to object
             self.activeQ.enter(obj)
             reactivate(obj,delay=0,prior=1)
```

(Here again I've omitted code, e.g. for the pre-emptable case, to simplify the exposition.)

A SimPy Source Code

Below is the SimPy source code. I've removed some of the triple-quoted comments at the beginning, and the test code at the end.

```
#!/usr/bin/env python
from SimPy.Lister import *
import heapq as hq
import types
import sys
import new
import random
import inspect

# $Revision: 1.1.1.75 $ $Date: 2007/12/18 13:30:47 $ kgm
"""Simulation 1.9 Implements SimPy Processes, Resources, Buffers, and the backbone simulation
scheduling by coroutine calls. Provides data collection through classes
Monitor and Tally.
Based on generators (Python 2.3 and later)
"""

# yield keywords
hold=1
passivate=2
request=3
release=4
waitevent=5
queueevent=6
waituntil=7
get=8
put=9

_endtime=0
_t=0
_e=None
_stop=True
_wustep=False #controls per event stepping for waituntil construct; not user API
try:
  True, False
except NameError:
  True, False = (1 == 1), (0 == 1)
condQ=[]
allMonitors=[]
allTallies=[]

def initialize():
    global _e,_t,_stop,condQ,allMonitors,allTallies
    _e=__Evlist()
    _t=0
    _stop=False
    condQ=[]
    allMonitors=[]
    allTallies=[]

def now():
    return _t

def stopSimulation():
    """Application function to stop simulation run"""
    global _stop
    _stop=True

def _startWUStepping():
    """Application function to start stepping through simulation for waituntil construct."""
    global _wustep
    _wustep=True

def _stopWUStepping():
    """Application function to stop stepping through simulation."""
    global _wustep
    _wustep=False

class Simerror(Exception):
    def __init__(self,value):
        self.value=value

    def __str__(self):
        return `self.value`

class FatalSimerror(Simerror):
    def __init__(self,value):
        Simerror.__init__(self,value)
        self.value=value

class Process(Lister):
    """Superclass of classes which may use generator functions"""
    def __init__(self,name="a_process"):
        #the reference to this Process instances single process (==generator)
        self._nextpoint=None
        self.name=name
        self._nextTime=None #next activation time
        self._remainService=0
        self._preempted=0
        self._priority={}
        self._getpriority={}
        self._putpriority={}
        self._terminated= False
        self._inInterrupt= False
        self.eventsFired=[] #which events process waited/queued for occurred

    def active(self):
        return self._nextTime <> None and not self._inInterrupt

    def passive(self):
        return self._nextTime is None and not self._terminated

    def terminated(self):
        return self._terminated

    def interrupted(self):
        return self._inInterrupt and not self._terminated

    def queuing(self,resource):
        return self in resource.waitQ

    def cancel(self,victim):
        """Application function to cancel all event notices for this Process
        instance;(should be all event notices for the _generator_)."""
        _e._unpost(whom=victim)

    def start(self,pem=None,at="undefined",delay="undefined",prior=False):
        """Activates PEM of this Process.
        p.start(p.pemname([args])[,{at= t |delay=period}][,prior=False]) or
        p.start([p.ACTIONS()][,{at= t |delay=period}][,prior=False]) (ACTIONS
                parameter optional)
        """
        if pem is None:
            try:
                pem=self.ACTIONS()
            except AttributeError:
                raise FatalSimerror\
                       ("Fatal SimPy error: no generator function to activate")
        else:
            pass
        if _e is None:
            raise FatalSimerror\
              ("Fatal SimPy error: simulation is not initialized"\
                                 "(call initialize() first)")
        if not (type(pem) == types.GeneratorType):
            raise FatalSimerror("Fatal SimPy error: activating function which"+
                           " is not a generator (contains no ’yield’)")
        if not self._terminated and not self._nextTime:
            #store generator reference in object; needed for reactivation
            self._nextpoint=pem
            if at=="undefined":
                at=_t
            if delay=="undefined":
                zeit=max(_t,at)
            else:
                zeit=max(_t,_t+delay)
            _e._post(what=self,at=zeit,prior=prior)

    def _hold(self,a):
        if len(a[0]) == 3:
            delay=abs(a[0][2])
        else:
            delay=0
        who=a[1]
        self.interruptLeft=delay
        self._inInterrupt=False
        self.interruptCause=None
        _e._post(what=who,at=_t+delay)
    def _passivate(self,a):
        a[0][1]._nextTime=None

    def interrupt(self,victim):
        """Application function to interrupt active processes"""
        # can’t interrupt terminated/passive/interrupted process
        if victim.active():
            victim.interruptCause=self  # self causes interrupt
            left=victim._nextTime-_t
            victim.interruptLeft=left   # time left in current ’hold’
            victim._inInterrupt=True
            reactivate(victim)
            return left
        else: #victim not active -- can’t interrupt
            return None

    def interruptReset(self):
        """
        Application function for an interrupt victim to get out of
        ’interrupted’ state.
        """
        self._inInterrupt= False

    def acquired(self,res):
        """Multi-functional test for reneging for ’request’ and ’get’:
        (1)If res of type Resource:
            Tests whether resource res was acquired when proces reactivated.
            If yes, the parallel wakeup process is killed.
            If not, process is removed from res.waitQ (reneging).
        (2)If res of type Store:
            Tests whether item(s) gotten from Store res.
            If yes, the parallel wakeup process is killed.
            If no, process is removed from res.getQ
        (3)If res of type Level:
            Tests whether units gotten from Level res.
            If yes, the parallel wakeup process is killed.
            If no, process is removed from res.getQ.
        """
        if isinstance(res,Resource):
            test=self in res.activeQ
            if test:
                self.cancel(self._holder)
            else:
                res.waitQ.remove(self)
                if res.monitored:
                    res.waitMon.observe(len(res.waitQ),t=now())
            return test
        elif isinstance(res,Store):
            test=len(self.got)
            if test:
                self.cancel(self._holder)
            else:
                res.getQ.remove(self)
                if res.monitored:
                    res.getQMon.observe(len(res.getQ),t=now())
            return test
        elif isinstance(res,Level):
            test=not (self.got is None)
            if test:
                self.cancel(self._holder)
            else:
                res.getQ.remove(self)
                if res.monitored:
                    res.getQMon.observe(len(res.getQ),t=now())
            return test

    def stored(self,buffer):
        """Test for reneging for ’yield put . . .’ compound statement (Level and
        Store. Returns True if not reneged.
        If self not in buffer.putQ, kill wakeup process, else take self out of
        buffer.putQ (reneged)"""
        test=self in buffer.putQ
        if test:    #reneged
            buffer.putQ.remove(self)
            if buffer.monitored:
                buffer.putQMon.observe(len(buffer.putQ),t=now())
        else:
            self.cancel(self._holder)
        return not test

def allEventNotices():
    """Returns string with eventlist as;
            t1: processname,processname2
            t2: processname4,processname5, . . .
            . . .  .
    """
    ret=""
    tempList=[]
    tempList[:]=_e.timestamps
    tempList.sort()
    # return only event notices which are not cancelled
    tempList=[[x[0],x[2].name] for x in tempList if not x[3]]
    tprev=-1
    for t in tempList:
        # if new time, new line
        if t[0]==tprev:
            # continue line
            ret+=",%s"%t[1]
        else:
            # new time
            if tprev==-1:
                ret="%s: %s"%(t[0],t[1])
            else:
                ret+="\n%s: %s"%(t[0],t[1])
            tprev=t[0]
    return ret+"\n"

def allEventTimes():
    """Returns list of all times for which events are scheduled.
    """
    r=[]
    r[:]=_e.timestamps
    r.sort()
    # return only event times of not cancelled event notices
    r1=[x[0] for x in r if not r[3]]
    tprev=-1
    ret=[]
    for t in r1:
        if t==tprev:
            #skip time, already in list
            pass
        else:
            ret.append(t)
            tprev=t
    return ret

class __Evlist(object):
    """Defines event list and operations on it"""
    def __init__(self):
        # always sorted list of events (sorted by time, priority)
        # make heapq
        self.timestamps = []
        self.sortpr=0

    def _post(self, what, at, prior=False):
        """Post an event notice for process what for time at"""
        # event notices are Process instances
        if at < _t:
            raise Simerror("Attempt to schedule event in the past")
        what._nextTime = at
        self.sortpr-=1
        if prior:
            # before all other event notices at this time
            # heappush with highest priority value so far (negative of monotonely increasing number)
            # store event notice in process instance
            what._rec=[at,self.sortpr,what,False]
            # make event list refer to it
            hq.heappush(self.timestamps,what._rec)
        else:
            # heappush with lowest priority
            # store event notice in process instance
            what._rec=[at,-self.sortpr,what,False]
            # make event list refer to it
            hq.heappush(self.timestamps,what._rec)

    def _unpost(self, whom):
        """
        Mark event notice for whom as cancelled if whom is a suspended process
        """
        if whom._nextTime is not None:  # check if whom was actually active
            whom._rec[3]=True ## Mark as cancelled
            whom._nextTime=None

    def _nextev(self):
        """Retrieve next event from event list"""
        global _t, _stop
        noActiveNotice=True
        ## Find next event notice which is not marked cancelled
        while noActiveNotice:
            if self.timestamps:
                 ## ignore priority value
                (_tnotice, p,nextEvent,cancelled) = hq.heappop(self.timestamps)
                noActiveNotice=cancelled
            else:
                raise Simerror("No more events at time %s" % _t)
        _t=_tnotice
        if _t > _endtime:
            _t = _endtime
            _stop = True
            return (None,)
        try:
            resultTuple = nextEvent._nextpoint.next()
        except StopIteration:
            nextEvent._nextpoint = None
            nextEvent._terminated = True
            nextEvent._nextTime = None
            resultTuple = None
        return (resultTuple, nextEvent)

    def _isEmpty(self):
        return not self.timestamps

    def _allEventNotices(self):
        """Returns string with eventlist as
                t1: [procname,procname2]
                t2: [procname4,procname5, . . . ]
                . . .  .
        """
        ret=""
        for t in self.timestamps:
            ret+="%s:%s\n"%(t[1]._nextTime, t[1].name)
        return ret[:-1]

    def _allEventTimes(self):
        """Returns list of all times for which events are scheduled.
        """
        return self.timestamps


def activate(obj,process,at="undefined",delay="undefined",prior=False):
    """Application function to activate passive process."""
    if _e is None:
        raise FatalSimerror\
       ("Fatal error: simulation is not initialized (call initialize() first)")
    if not (type(process) == types.GeneratorType):
        raise FatalSimerror("Activating function which"+
                       " is not a generator (contains no ’yield’)")
    if not obj._terminated and not obj._nextTime:
        #store generator reference in object; needed for reactivation
        obj._nextpoint=process
        if at=="undefined":
            at=_t
        if delay=="undefined":
            zeit=max(_t,at)
        else:
            zeit=max(_t,_t+delay)
        _e._post(obj,at=zeit,prior=prior)

def reactivate(obj,at="undefined",delay="undefined",prior=False):
    """Application function to reactivate a process which is active,
    suspended or passive."""
    # Object may be active, suspended or passive
    if not obj._terminated:
        a=Process("SimPysystem")
        a.cancel(obj)
        # object now passive
        if at=="undefined":
            at=_t
        if delay=="undefined":
            zeit=max(_t,at)
        else:
            zeit=max(_t,_t+delay)
        _e._post(obj,at=zeit,prior=prior)

class Histogram(list):
    """ A histogram gathering and sampling class"""

    def __init__(self,name = ’’,low=0.0,high=100.0,nbins=10):
        list.__init__(self)
        self.name  = name
        self.low   = float(low)
        self.high  = float(high)
        self.nbins = nbins
        self.binsize=(self.high-self.low)/nbins
        self._nrObs=0
        self._sum=0
        self[:] =[[low+(i-1)*self.binsize,0] for i in range(self.nbins+2)]

    def addIn(self,y):
        """ add a value into the correct bin"""
        self._nrObs+=1
        self._sum+=y
        b = int((y-self.low+self.binsize)/self.binsize)
        if b < 0: b = 0
        if b > self.nbins+1: b = self.nbins+1
        assert 0 <= b <=self.nbins+1,’Histogram.addIn: b out of range: %s’%b
        self[b][1]+=1

    def __str__(self):
        histo=self
        ylab="value"
        nrObs=self._nrObs
        width=len(str(nrObs))
        res=[]
        res.append("<Histogram %s:"%self.name)
        res.append("\nNumber of observations: %s"%nrObs)
        if nrObs:
            su=self._sum
            cum=histo[0][1]
            fmt="%s"
            line="\n%s <= %s < %s: %s (cum: %s/%s%s)"\
                 %(fmt,"%s",fmt,"%s","%s","%5.1f","%s")
            line1="\n%s%s < %s: %s (cum: %s/%s%s)"\
                 %("%s","%s",fmt,"%s","%s","%5.1f","%s")
            l1width=len(("%s <= "%fmt)%histo[1][0])
            res.append(line1\
                       %(" "*l1width,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
                         str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                      )
            for i in range(1,len(histo)-1):
                cum+=histo[i][1]
                res.append(line\
                       %(histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
                         str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                          )
            cum+=histo[-1][1]
            linen="\n%s <= %s %s : %s (cum: %s/%s%s)"\
                  %(fmt,"%s","%s","%s","%s","%5.1f","%s")
            lnwidth=len(("<%s"%fmt)%histo[1][0])
            res.append(linen\
                       %(histo[-1][0],ylab," "*lnwidth,str(histo[-1][1]).rjust(width),\
                       str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                       )
        res.append("\n>")
        return " ".join(res)

def startCollection(when=0.0,monitors=None,tallies=None):
    """Starts data collection of all designated Monitor and Tally objects
    (default=all) at time ’when’.
    """
    class Starter(Process):
        def collect(self,monitors,tallies):
            for m in monitors:
                print m.name
                m.reset()
            for t in tallies:
                t.reset()
            yield hold,self
    if monitors is None:
        monitors=allMonitors
    if tallies is None:
        tallies=allTallies
    s=Starter()
    activate(s,s.collect(monitors=monitors,tallies=tallies),at=when)

class Monitor(list):
    """ Monitored variables

    A Class for monitored variables, that is, variables that allow one
    to gather simple statistics.  A Monitor is a subclass of list and
    list operations can be performed on it. An object is established
    using m= Monitor(name = ’..’). It can be given a
    unique name for use in debugging and in tracing and ylab and tlab
    strings for labelling graphs.
    """
    def __init__(self,name=’a_Monitor’,ylab=’y’,tlab=’t’):
        list.__init__(self)
        self.startTime = 0.0
        self.name = name
        self.ylab = ylab
        self.tlab = tlab
        allMonitors.append(self)

    def setHistogram(self,name = ’’,low=0.0,high=100.0,nbins=10):
        """Sets histogram parameters.
        Must be called before call to getHistogram"""
        if name==’’:
            histname=self.name
        else:
            histname=name
        self.histo=Histogram(name=histname,low=low,high=high,nbins=nbins)

    def observe(self,y,t=None):
        """record y and t"""
        if t is  None: t = now()
        self.append([t,y])

    def tally(self,y):
        """ deprecated: tally for backward compatibility"""
        self.observe(y,0)

    def accum(self,y,t=None):
        """ deprecated:  accum for backward compatibility"""
        self.observe(y,t)

    def reset(self,t=None):
        """reset the sums and counts for the monitored variable """
        self[:]=[]
        if t is None: t = now()
        self.startTime = t

    def tseries(self):
        """ the series of measured times"""
        return list(zip(*self)[0])

    def yseries(self):
        """ the series of measured values"""
        return list(zip(*self)[1])

    def count(self):
        """ deprecated: the number of observations made """
        return self.__len__()

    def total(self):
        """ the sum of the y"""
        if self.__len__()==0:  return 0
        else:
            sum = 0.0
            for i in range(self.__len__()):
                sum += self[i][1]
            return sum # replace by sum() later

    def mean(self):
        """ the simple average of the monitored variable"""
        try: return 1.0*self.total()/self.__len__()
        except:  print ’SimPy: No observations  for mean’

    def var(self):
        """ the sample variance of the monitored variable """
        n = len(self)
        tot = self.total()
        ssq=0.0
        ##yy = self.yseries()
        for i in range(self.__len__()):
            ssq += self[i][1]**2 # replace by sum() eventually
        try: return (ssq - float(tot*tot)/n)/n
        except: print ’SimPy: No observations for sample variance’

    def timeAverage(self,t=None):
        """ the time-weighted average of the monitored variable.

            If t is used it is assumed to be the current time,
            otherwise t =  now()
        """
        N = self.__len__()
        if N  == 0:
            print ’SimPy: No observations for timeAverage’
            return None

        if t is None: t = now()
        sum = 0.0
        tlast = self.startTime
        #print ’DEBUG: timave ’,t,tlast
        ylast = 0.0
        for i in range(N):
            ti,yi = self[i]
            sum += ylast*(ti-tlast)
            tlast = ti
            ylast = yi
        sum += ylast*(t-tlast)
        T = t - self.startTime
        if T == 0:
             print ’SimPy: No elapsed time for timeAverage’
             return None
        #print ’DEBUG: timave ’,sum,t,T
        return sum/float(T)

    def timeVariance(self,t=None):
        """ the time-weighted Variance of the monitored variable.

            If t is used it is assumed to be the current time,
            otherwise t =  now()
        """
        N = self.__len__()
        if N  == 0:
            print ’SimPy: No observations for timeVariance’
            return None
        if t is None: t = now()
        sm = 0.0
        ssq = 0.0
        tlast = self.startTime
        # print ’DEBUG: 1 twVar ’,t,tlast
        ylast = 0.0
        for i in range(N):
            ti,yi = self[i]
            sm  += ylast*(ti-tlast)
            ssq += ylast*ylast*(ti-tlast)
            tlast = ti
            ylast = yi
        sm  += ylast*(t-tlast)
        ssq += ylast*ylast*(t-tlast)
        T = t - self.startTime
        if T == 0:
             print ’SimPy: No elapsed time for timeVariance’
             return None
        mn = sm/float(T)
        # print ’DEBUG: 2 twVar ’,ssq,t,T
        return ssq/float(T) - mn*mn


    def histogram(self,low=0.0,high=100.0,nbins=10):
        """ A histogram of the monitored y data values.
        """
        h = Histogram(name=self.name,low=low,high=high,nbins=nbins)
        ys = self.yseries()
        for y in ys: h.addIn(y)
        return h

    def getHistogram(self):
        """Returns a histogram based on the parameters provided in
        preceding call to setHistogram.
        """
        ys = self.yseries()
        h=self.histo
        for y in ys: h.addIn(y)
        return h

    def printHistogram(self,fmt="%s"):
        """Returns formatted frequency distribution table string from Monitor.
        Precondition: setHistogram must have been called.
        fmt==format of bin range values
        """
        try:
            histo=self.getHistogram()
        except:
            raise FatalSimerror("histogramTable: call setHistogram first"\
                                " for Monitor %s"%self.name)
        ylab=self.ylab
        nrObs=self.count()
        width=len(str(nrObs))
        res=[]
        res.append("\nHistogram for %s:"%histo.name)
        res.append("\nNumber of observations: %s"%nrObs)
        su=sum(self.yseries())
        cum=histo[0][1]
        line="\n%s <= %s < %s: %s (cum: %s/%s%s)"\
             %(fmt,"%s",fmt,"%s","%s","%5.1f","%s")
        line1="\n%s%s < %s: %s (cum: %s/%s%s)"\
             %("%s","%s",fmt,"%s","%s","%5.1f","%s")
        l1width=len(("%s <= "%fmt)%histo[1][0])
        res.append(line1\
                   %(" "*l1width,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
                     str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                  )
        for i in range(1,len(histo)-1):
            cum+=histo[i][1]
            res.append(line\
                   %(histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
                     str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                      )
        cum+=histo[-1][1]
        linen="\n%s <= %s %s : %s (cum: %s/%s%s)"\
              %(fmt,"%s","%s","%s","%s","%5.1f","%s")
        lnwidth=len(("<%s"%fmt)%histo[1][0])
        res.append(linen\
                   %(histo[-1][0],ylab," "*lnwidth,str(histo[-1][1]).rjust(width),\
                   str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                   )
        return " ".join(res)

class Tally:
    def __init__(self, name="a_Tally", ylab="y",tlab="t"):
        self.name = name
        self.ylab = ylab
        self.tlab = tlab
        self.reset()
        self.startTime = 0.0
        self.histo = None
        self.sum = 0.0
        self._sum_of_squares = 0
        self._integral = 0.0    # time-weighted sum
        self._integral2 = 0.0   # time-weighted sum of squares
        allTallies.append(self)

    def setHistogram(self,name = ’’,low=0.0,high=100.0,nbins=10):
        """Sets histogram parameters.
        Must be called to prior to observations initiate data collection
        for histogram.
        """
        if name==’’:
            hname=self.name
        else:
            hname=name
        self.histo=Histogram(name=hname,low=low,high=high,nbins=nbins)

    def observe(self, y, t=None):
        if t is None:
            t = now()
        self._integral += (t - self._last_timestamp) * self._last_observation
        yy =  self._last_observation* self._last_observation
        self._integral2 += (t - self._last_timestamp) * yy
        self._last_timestamp = t
        self._last_observation = y
        self._total += y
        self._count += 1
        self._sum += y
        self._sum_of_squares += y * y
        if self.histo:
            self.histo.addIn(y)

    def reset(self, t=None):
        if t is None:
            t = now()
        self.startTime = t
        self._last_timestamp = t
        self._last_observation = 0.0
        self._count = 0
        self._total = 0.0
        self._integral = 0.0
        self._integral2 = 0.0
        self._sum = 0.0
        self._sum_of_squares = 0.0

    def count(self):
        return self._count

    def total(self):
        return self._total

    def mean(self):
        return 1.0 * self._total / self._count

    def timeAverage(self,t=None):
        if t is None:
            t=now()
        integ=self._integral+(t - self._last_timestamp) * self._last_observation
        if (t > self.startTime):
            return 1.0 * integ/(t - self.startTime)
        else:
            print ’SimPy: No elapsed time for timeAverage’
            return None

    def var(self):
        return 1.0 * (self._sum_of_squares - (1.0 * (self._sum * self._sum)\
               / self._count)) / (self._count)

    def timeVariance(self,t=None):
        """ the time-weighted Variance of the Tallied variable.

            If t is used it is assumed to be the current time,
            otherwise t =  now()
        """
        if t is None:
            t=now()
        twAve = self.timeAverage(t)
        #print ’Tally timeVariance DEBUG: twave:’, twAve
        last =  self._last_observation
        twinteg2=self._integral2+(t - self._last_timestamp) * last * last
        #print ’Tally timeVariance DEBUG:tinteg2:’, twinteg2
        if (t > self.startTime):
            return 1.0 * twinteg2/(t - self.startTime) - twAve*twAve
        else:
            print ’SimPy: No elapsed time for timeVariance’
            return None



    def __len__(self):
        return self._count

    def __eq__(self, l):
        return len(l) == self._count

    def getHistogram(self):
        return self.histo

    def printHistogram(self,fmt="%s"):
        """Returns formatted frequency distribution table string from Tally.
        Precondition: setHistogram must have been called.
        fmt==format of bin range values
        """
        try:
            histo=self.getHistogram()
        except:
            raise FatalSimerror("histogramTable: call setHistogram first"\
                                " for Tally %s"%self.name)
        ylab=self.ylab
        nrObs=self.count()
        width=len(str(nrObs))
        res=[]
        res.append("\nHistogram for %s:"%histo.name)
        res.append("\nNumber of observations: %s"%nrObs)
        su=self.total()
        cum=histo[0][1]
        line="\n%s <= %s < %s: %s (cum: %s/%s%s)"\
             %(fmt,"%s",fmt,"%s","%s","%5.1f","%s")
        line1="\n%s%s < %s: %s (cum: %s/%s%s)"\
             %("%s","%s",fmt,"%s","%s","%5.1f","%s")
        l1width=len(("%s <= "%fmt)%histo[1][0])
        res.append(line1\
                   %(" "*l1width,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
                     str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                  )
        for i in range(1,len(histo)-1):
            cum+=histo[i][1]
            res.append(line\
                   %(histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
                     str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                      )
        cum+=histo[-1][1]
        linen="\n%s <= %s %s : %s (cum: %s/%s%s)"\
              %(fmt,"%s","%s","%s","%s","%5.1f","%s")
        lnwidth=len(("<%s"%fmt)%histo[1][0])
        res.append(linen\
                   %(histo[-1][0],ylab," "*lnwidth,str(histo[-1][1]).rjust(width),\
                   str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
                   )
        return " ".join(res)

class Queue(list):
    def __init__(self,res,moni):
        if not moni is None: #moni==[]:
            self.monit=True # True if a type of Monitor/Tally attached
        else:
            self.monit=False
        self.moni=moni # The Monitor/Tally
        self.resource=res # the resource/buffer this queue belongs to

    def enter(self,obj):
        pass

    def leave(self):
        pass

    def takeout(self,obj):
        self.remove(obj)
        if self.monit:
            self.moni.observe(len(self),t=now())

class FIFO(Queue):
    def __init__(self,res,moni):
        Queue.__init__(self,res,moni)

    def enter(self,obj):
        self.append(obj)
        if self.monit:
            self.moni.observe(len(self),t=now())

    def enterGet(self,obj):
        self.enter(obj)

    def enterPut(self,obj):
        self.enter(obj)

    def leave(self):
        a= self.pop(0)
        if self.monit:
            self.moni.observe(len(self),t=now())
        return a

class PriorityQ(FIFO):
    """Queue is always ordered according to priority.
    Higher value of priority attribute == higher priority.
    """
    def __init__(self,res,moni):
        FIFO.__init__(self,res,moni)

    def enter(self,obj):
        """Handles request queue for Resource"""
        if len(self):
            ix=self.resource
            if self[-1]._priority[ix] >= obj._priority[ix]:
                self.append(obj)
            else:
                z=0
                while self[z]._priority[ix] >= obj._priority[ix]:
                    z += 1
                self.insert(z,obj)
        else:
            self.append(obj)
        if self.monit:
            self.moni.observe(len(self),t=now())

    def enterGet(self,obj):
        """Handles getQ in Buffer"""
        if len(self):
            ix=self.resource
            #print "priority:",[x._priority[ix] for x in self]
            if self[-1]._getpriority[ix] >= obj._getpriority[ix]:
                self.append(obj)
            else:
                z=0
                while self[z]._getpriority[ix] >= obj._getpriority[ix]:
                    z += 1
                self.insert(z,obj)
        else:
            self.append(obj)
        if self.monit:
            self.moni.observe(len(self),t=now())

    def enterPut(self,obj):
        """Handles putQ in Buffer"""
        if len(self):
            ix=self.resource
            #print "priority:",[x._priority[ix] for x in self]
            if self[-1]._putpriority[ix] >= obj._putpriority[ix]:
                self.append(obj)
            else:
                z=0
                while self[z]._putpriority[ix] >= obj._putpriority[ix]:
                    z += 1
                self.insert(z,obj)
        else:
            self.append(obj)
        if self.monit:
            self.moni.observe(len(self),t=now())

class Resource(Lister):
    """Models shared, limited capacity resources with queuing;
    FIFO is default queuing discipline.
    """

    def __init__(self,capacity=1,name="a_resource",unitName="units",
                 qType=FIFO,preemptable=0,monitored=False,monitorType=Monitor):
        """
        monitorType={Monitor(default)|Tally}
        """
        self.name=name          # resource name
        self.capacity=capacity  # resource units in this resource
        self.unitName=unitName  # type name of resource units
        self.n=capacity         # uncommitted resource units
        self.monitored=monitored

        if self.monitored:           # Monitor waitQ, activeQ
            self.actMon=monitorType(name="Active Queue Monitor %s"%self.name,
                                 ylab="nr in queue",tlab="time")
            monact=self.actMon
            self.waitMon=monitorType(name="Wait Queue Monitor %s"%self.name,
                                 ylab="nr in queue",tlab="time")
            monwait=self.waitMon
        else:
            monwait=None
            monact=None
        self.waitQ=qType(self,monwait)
        self.preemptable=preemptable
        self.activeQ=qType(self,monact)
        self.priority_default=0

    def _request(self,arg):
        """Process request event for this resource"""
        obj=arg[1]
        if len(arg[0]) == 4:        # yield request,self,resource,priority
            obj._priority[self]=arg[0][3]
        else:                       # yield request,self,resource
            obj._priority[self]=self.priority_default
        if self.preemptable and self.n == 0: # No free resource
            # test for preemption condition
            preempt=obj._priority[self] > self.activeQ[-1]._priority[self]
            # If yes:
            if preempt:
                z=self.activeQ[-1]
                # suspend lowest priority process being served
                ##suspended = z
                # record remaining service time
                z._remainService = z._nextTime - _t
                Process().cancel(z)
                # remove from activeQ
                self.activeQ.remove(z)
                # put into front of waitQ
                self.waitQ.insert(0,z)
                # if self is monitored, update waitQ monitor
                if self.monitored:
                    self.waitMon.observe(len(self.waitQ),now())
                # record that it has been preempted
                z._preempted = 1
                # passivate re-queued process
                z._nextTime=None
                # assign resource unit to preemptor
                self.activeQ.enter(obj)
                # post event notice for preempting process
                _e._post(obj,at=_t,prior=1)
            else:
                self.waitQ.enter(obj)
                # passivate queuing process
                obj._nextTime=None
        else: # treat non-preemption case
            if self.n == 0:
                self.waitQ.enter(obj)
                # passivate queuing process
                obj._nextTime=None
            else:
                self.n -= 1
                self.activeQ.enter(obj)
                _e._post(obj,at=_t,prior=1)

    def _release(self,arg):
        """Process release request for this resource"""
        self.n += 1
        self.activeQ.remove(arg[1])
        if self.monitored:
            self.actMon.observe(len(self.activeQ),t=now())
        #reactivate first waiting requestor if any; assign Resource to it
        if self.waitQ:
            obj=self.waitQ.leave()
            self.n -= 1             #assign 1 resource unit to object
            self.activeQ.enter(obj)
            # if resource preemptable:
            if self.preemptable:
                # if object had been preempted:
                if obj._preempted:
                    obj._preempted = 0
                    # reactivate object delay= remaining service time
                    reactivate(obj,delay=obj._remainService)
                # else reactivate right away
                else:
                    reactivate(obj,delay=0,prior=1)
            # else:
            else:
                reactivate(obj,delay=0,prior=1)
        _e._post(arg[1],at=_t,prior=1)

class Buffer(Lister):
    """Abstract class for buffers
    Blocks a process when a put would cause buffer overflow or a get would cause
    buffer underflow.
    Default queuing discipline for blocked processes is FIFO."""

    priorityDefault=0
    def __init__(self,name=None,capacity="unbounded",unitName="units",
                putQType=FIFO,getQType=FIFO,
                monitored=False,monitorType=Monitor,initialBuffered=None):
        if capacity=="unbounded": capacity=sys.maxint
        self.capacity=capacity
        self.name=name
        self.putQType=putQType
        self.getQType=getQType
        self.monitored=monitored
        self.initialBuffered=initialBuffered
        self.unitName=unitName
        if self.monitored:
            ## monitor for Producer processes' queue
            self.putQMon=monitorType(name="Producer Queue Monitor %s"%self.name,
                                    ylab="nr in queue",tlab="time")
            ## monitor for Consumer processes' queue
            self.getQMon=monitorType(name="Consumer Queue Monitor %s"%self.name,
                                    ylab="nr in queue",tlab="time")
            ## monitor for nr items in buffer
            self.bufferMon=monitorType(name="Buffer Monitor %s"%self.name,
                                    ylab="nr in buffer",tlab="time")
        else:
            self.putQMon=None
            self.getQMon=None
            self.bufferMon=None
        self.putQ=self.putQType(res=self,moni=self.putQMon)
        self.getQ=self.getQType(res=self,moni=self.getQMon)
        if self.monitored:
            self.putQMon.observe(y=len(self.putQ),t=now())
            self.getQMon.observe(y=len(self.getQ),t=now())
        self._putpriority={}
        self._getpriority={}

        def _put(self):
            pass
        def _get(self):
            pass

class Level(Buffer):
    """Models buffers for processes putting/getting un-distinguishable items.
    """
    def getamount(self):
        return self.nrBuffered

    def gettheBuffer(self):
        return self.nrBuffered

    theBuffer=property(gettheBuffer)

    def __init__(self,**pars):
        Buffer.__init__(self,**pars)
        if self.name is None:
            self.name="a_level"   ## default name

        if (type(self.capacity)!=type(1.0) and\
                type(self.capacity)!=type(1)) or\
                self.capacity<0:
                raise FatalSimerror\
                    ("Level: capacity parameter not a positive number: %s"\
                    %self.initialBuffered)

        if type(self.initialBuffered)==type(1.0) or\
                type(self.initialBuffered)==type(1):
            if self.initialBuffered>self.capacity:
                raise FatalSimerror("initialBuffered exceeds capacity")
            if self.initialBuffered>=0:
                self.nrBuffered=self.initialBuffered ## nr items initially in buffer
                                        ## buffer is just a counter (int type)
            else:
                raise FatalSimerror\
                ("initialBuffered param of Level negative: %s"\
                %self.initialBuffered)
        elif self.initialBuffered is None:
            self.initialBuffered=0
            self.nrBuffered=0
        else:
            raise FatalSimerror\
                ("Level: wrong type of initialBuffered (parameter=%s)"\
                %self.initialBuffered)
        if self.monitored:
            self.bufferMon.observe(y=self.amount,t=now())
    amount=property(getamount)

    def _put(self,arg):
        """Handles put requests for Level instances"""
        obj=arg[1]
        if len(arg[0]) == 5:        # yield put,self,buff,whattoput,priority
            obj._putpriority[self]=arg[0][4]
            whatToPut=arg[0][3]
        elif len(arg[0]) == 4:      # yield get,self,buff,whattoput
            obj._putpriority[self]=Buffer.priorityDefault #default
            whatToPut=arg[0][3]
        else:                       # yield get,self,buff
            obj._putpriority[self]=Buffer.priorityDefault #default
            whatToPut=1
        if type(whatToPut)!=type(1) and type(whatToPut)!=type(1.0):
            raise FatalSimerror("Level: put parameter not a number")
        if not whatToPut>=0.0:
            raise FatalSimerror("Level: put parameter not positive number")
        whatToPutNr=whatToPut
        if whatToPutNr+self.amount>self.capacity:
            obj._nextTime=None      #passivate put requestor
            obj._whatToPut=whatToPutNr
            self.putQ.enterPut(obj)    #and queue, with size of put
        else:
            self.nrBuffered+=whatToPutNr
            if self.monitored:
                self.bufferMon.observe(y=self.amount,t=now())
            # service any getters waiting
            # service in queue-order; do not serve second in queue before first
            # has been served
            while len(self.getQ) and self.amount>0:
                proc=self.getQ[0]
                if proc._nrToGet<=self.amount:
                    proc.got=proc._nrToGet
                    self.nrBuffered-=proc.got
                    if self.monitored:
                        self.bufferMon.observe(y=self.amount,t=now())
                    self.getQ.takeout(proc) # get requestor’s record out of queue
                    _e._post(proc,at=_t) # continue a blocked get requestor
                else:
                    break
            _e._post(obj,at=_t,prior=1) # continue the put requestor

    def _get(self,arg):
        """Handles get requests for Level instances"""
        obj=arg[1]
        obj.got=None
        if len(arg[0]) == 5:        # yield get,self,buff,whattoget,priority
            obj._getpriority[self]=arg[0][4]
            nrToGet=arg[0][3]
        elif len(arg[0]) == 4:      # yield get,self,buff,whattoget
            obj._getpriority[self]=Buffer.priorityDefault #default
            nrToGet=arg[0][3]
        else:                       # yield get,self,buff
            obj._getpriority[self]=Buffer.priorityDefault
            nrToGet=1
        if type(nrToGet)!=type(1.0) and type(nrToGet)!=type(1):
            raise FatalSimerror\
                ("Level: get parameter not a number: %s"%nrToGet)
        if nrToGet<0:
            raise FatalSimerror\
                ("Level: get parameter not positive number: %s"%nrToGet)
        if self.amount < nrToGet:
            obj._nrToGet=nrToGet
            self.getQ.enterGet(obj)
            # passivate queuing process
            obj._nextTime=None
        else:
            obj.got=nrToGet
            self.nrBuffered-=nrToGet
            if self.monitored:
                self.bufferMon.observe(y=self.amount,t=now())
            _e._post(obj,at=_t,prior=1)
            # reactivate any put requestors for which space is now available
            # service in queue-order; do not serve second in queue before first
            # has been served
            while len(self.putQ): #test for queued producers
                proc=self.putQ[0]
                if proc._whatToPut+self.amount<=self.capacity:
                    self.nrBuffered+=proc._whatToPut
                    if self.monitored:
                        self.bufferMon.observe(y=self.amount,t=now())
                    self.putQ.takeout(proc)#requestor’s record out of queue
                    _e._post(proc,at=_t) # continue a blocked put requestor
                else:
                    break

class Store(Buffer):
    """Models buffers for processes coupled by putting/getting distinguishable
    items.
    Blocks a process when a put would cause buffer overflow or a get would cause
    buffer underflow.
    Default queuing discipline for blocked processes is priority FIFO.
    """
    def getnrBuffered(self):
        return len(self.theBuffer)
    nrBuffered=property(getnrBuffered)

    def getbuffered(self):
        return self.theBuffer
    buffered=property(getbuffered)

    def __init__(self,**pars):
        Buffer.__init__(self,**pars)
        self.theBuffer=[]
        if self.name is None:
            self.name="a_store" ## default name
        if type(self.capacity)!=type(1) or self.capacity<=0:
            raise FatalSimerror\
                ("Store: capacity parameter not a positive integer > 0: %s"\
                    %self.initialBuffered)
        if type(self.initialBuffered)==type([]):
            if len(self.initialBuffered)>self.capacity:
                raise FatalSimerror("initialBuffered exceeds capacity")
            else:
                self.theBuffer[:]=self.initialBuffered##buffer==list of objects
        elif self.initialBuffered is None:
            self.theBuffer=[]
        else:
            raise FatalSimerror\
                ("Store: initialBuffered not a list")
        if self.monitored:
            self.bufferMon.observe(y=self.nrBuffered,t=now())
        self._sort=None



    def addSort(self,sortFunc):
        """Adds buffer sorting to this instance of Store. It maintains
        theBuffer sorted by the sortAttr attribute of the objects in the
        buffer.
        The user-provided ’sortFunc’ must look like this:

        def mySort(self,par):
            tmplist=[(x.sortAttr,x) for x in par]
            tmplist.sort()
            return [x for (key,x) in tmplist]

        """

        self._sort=new.instancemethod(sortFunc,self,self.__class__)
        self.theBuffer=self._sort(self.theBuffer)

    def _put(self,arg):
        """Handles put requests for Store instances"""
        obj=arg[1]
        if len(arg[0]) == 5:        # yield put,self,buff,whattoput,priority
            obj._putpriority[self]=arg[0][4]
            whatToPut=arg[0][3]
        elif len(arg[0]) == 4:      # yield put,self,buff,whattoput
            obj._putpriority[self]=Buffer.priorityDefault #default
            whatToPut=arg[0][3]
        else:                       # error, whattoput missing
            raise FatalSimerror("Item to put missing in yield put stmt")
        if type(whatToPut)!=type([]):
            raise FatalSimerror("put parameter is not a list")
        whatToPutNr=len(whatToPut)
        if whatToPutNr+self.nrBuffered>self.capacity:
            obj._nextTime=None      #passivate put requestor
            obj._whatToPut=whatToPut
            self.putQ.enterPut(obj) #and queue, with items to put
        else:
            self.theBuffer.extend(whatToPut)
            if not(self._sort is None):
                self.theBuffer=self._sort(self.theBuffer)
            if self.monitored:
                self.bufferMon.observe(y=self.nrBuffered,t=now())

            # service any waiting getters
            # service in queue order: do not serve second in queue before first
            # has been served
            while self.nrBuffered>0 and len(self.getQ):
                proc=self.getQ[0]
                if inspect.isfunction(proc._nrToGet):
                    movCand=proc._nrToGet(self.theBuffer) #predicate parameter
                    if movCand:
                        proc.got=movCand[:]
                        for i in movCand:
                            self.theBuffer.remove(i)
                        self.getQ.takeout(proc)
                        if self.monitored:
                            self.bufferMon.observe(y=self.nrBuffered,t=now())
                        _e._post(what=proc,at=_t) # continue a blocked get requestor
                    else:
                        break
                else: #numerical parameter
                    if proc._nrToGet<=self.nrBuffered:
                        nrToGet=proc._nrToGet
                        proc.got=[]
                        proc.got[:]=self.theBuffer[0:nrToGet]
                        self.theBuffer[:]=self.theBuffer[nrToGet:]
                        if self.monitored:
                            self.bufferMon.observe(y=self.nrBuffered,t=now())
                        # take this get requestor’s record out of queue:
                        self.getQ.takeout(proc)
                        _e._post(what=proc,at=_t) # continue a blocked get requestor
                    else:
                        break

            _e._post(what=obj,at=_t,prior=1) # continue the put requestor

    def _get(self,arg):
        """Handles get requests"""
        filtfunc=None
        obj=arg[1]
        obj.got=[]                  # the list of items retrieved by ’get’
        if len(arg[0]) == 5:        # yield get,self,buff,whattoget,priority
            obj._getpriority[self]=arg[0][4]
            if inspect.isfunction(arg[0][3]):
                filtfunc=arg[0][3]
            else:
                nrToGet=arg[0][3]
        elif len(arg[0]) == 4:      # yield get,self,buff,whattoget
            obj._getpriority[self]=Buffer.priorityDefault #default
            if inspect.isfunction(arg[0][3]):
                filtfunc=arg[0][3]
            else:
                nrToGet=arg[0][3]
        else:                       # yield get,self,buff
            obj._getpriority[self]=Buffer.priorityDefault
            nrToGet=1
        if not filtfunc: #number specifies nr items to get
            if nrToGet<0:
                raise FatalSimerror\
                    ("Store: get parameter not positive number: %s"%nrToGet)
            if self.nrBuffered < nrToGet:
                obj._nrToGet=nrToGet
                self.getQ.enterGet(obj)
                # passivate/block queuing ’get’ process
                obj._nextTime=None
            else:
                for i in range(nrToGet):
                    obj.got.append(self.theBuffer.pop(0)) # move items from
                                                # buffer to requesting process
                if self.monitored:
                    self.bufferMon.observe(y=self.nrBuffered,t=now())
                _e._post(obj,at=_t,prior=1)
                # reactivate any put requestors for which space is now available
                # serve in queue order: do not serve second in queue before first
                # has been served
                while len(self.putQ):
                    proc=self.putQ[0]
                    if len(proc._whatToPut)+self.nrBuffered<=self.capacity:
                        for i in proc._whatToPut:
                            self.theBuffer.append(i) #move items to buffer
                        if not(self._sort is None):
                            self.theBuffer=self._sort(self.theBuffer)
                        if self.monitored:
                            self.bufferMon.observe(y=self.nrBuffered,t=now())
                        self.putQ.takeout(proc) # dequeue requestor’s record
                        _e._post(proc,at=_t) # continue a blocked put requestor
                    else:
                        break
        else: # items to get determined by filtfunc
            movCand=filtfunc(self.theBuffer)
            if movCand: # get succeded
                _e._post(obj,at=_t,prior=1)
                obj.got=movCand[:]
                for item in movCand:
                    self.theBuffer.remove(item)
                if self.monitored:
                    self.bufferMon.observe(y=self.nrBuffered,t=now())
                # reactivate any put requestors for which space is now available
                # serve in queue order: do not serve second in queue before first
                # has been served
                while len(self.putQ):
                    proc=self.putQ[0]
                    if len(proc._whatToPut)+self.nrBuffered<=self.capacity:
                        for i in proc._whatToPut:
                            self.theBuffer.append(i) #move items to buffer
                        if not(self._sort is None):
                            self.theBuffer=self._sort(self.theBuffer)
                        if self.monitored:
                            self.bufferMon.observe(y=self.nrBuffered,t=now())
                        self.putQ.takeout(proc) # dequeue requestor’s record
                        _e._post(proc,at=_t) # continue a blocked put requestor
                    else:
                        break
            else: # get did not succeed, block
                obj._nrToGet=filtfunc
                self.getQ.enterGet(obj)
                # passivate/block queuing ’get’ process
                obj._nextTime=None

class SimEvent(Lister):
    """Supports one-shot signalling between processes. All processes waiting for an event to occur
    get activated when its occurrence is signalled. From the processes queuing for an event, only
    the first gets activated.
    """
    def __init__(self,name="a_SimEvent"):
        self.name=name
        self.waits=[]
        self.queues=[]
        self.occurred=False
        self.signalparam=None

    def signal(self,param=None):
        """Produces a signal to self;
        Fires this event (makes it occur).
        Reactivates ALL processes waiting for this event. (Cleanup waits lists
        of other events if wait was for an event-group (OR).)
        Reactivates the first process for which event(s) it is queuing for
        have fired. (Cleanup queues of other events if wait was for an event-group (OR).)
        """
        self.signalparam=param
        if not self.waits and not self.queues:
            self.occurred=True
        else:
            #reactivate all waiting processes
            for p in self.waits:
                p[0].eventsFired.append(self)
                reactivate(p[0],prior=True)
                #delete waits entries for this process in other events
                for ev in p[1]:
                    if ev!=self:
                        if ev.occurred:
                            p[0].eventsFired.append(ev)
                        for iev in ev.waits:
                            if iev[0]==p[0]:
                                ev.waits.remove(iev)
                                break
            self.waits=[]
            if self.queues:
                proc=self.queues.pop(0)[0]
                proc.eventsFired.append(self)
                reactivate(proc)

    def _wait(self,par):
        """Consumes a signal if it has occurred, otherwise process ’proc’
        waits for this event.
        """
        proc=par[0][1] #the process issuing the yield waitevent command
        proc.eventsFired=[]
        if not self.occurred:
            self.waits.append([proc,[self]])
            proc._nextTime=None #passivate calling process
        else:
            proc.eventsFired.append(self)
            self.occurred=False
            _e._post(proc,at=_t,prior=1)

    def _waitOR(self,par):
        """Handles waiting for an OR of events in a tuple/list.
        """
        proc=par[0][1]
        evlist=par[0][2]
        proc.eventsFired=[]
        anyoccur=False
        for ev in evlist:
            if ev.occurred:
                anyoccur=True
                proc.eventsFired.append(ev)
                ev.occurred=False
        if anyoccur: #at least one event has fired; continue process
            _e._post(proc,at=_t,prior=1)

        else: #no event in list has fired, enter process in all ’waits’ lists
            proc.eventsFired=[]
            proc._nextTime=None #passivate calling process
            for ev in evlist:
                ev.waits.append([proc,evlist])

    def _queue(self,par):
        """Consumes a signal if it has occurred, otherwise process ’proc’
        queues for this event.
        """
        proc=par[0][1] #the process issuing the yield queueevent command
        proc.eventsFired=[]
        if not self.occurred:
            self.queues.append([proc,[self]])
            proc._nextTime=None #passivate calling process
        else:
            proc.eventsFired.append(self)
            self.occurred=False
            _e._post(proc,at=_t,prior=1)

    def _queueOR(self,par):
        """Handles queueing for an OR of events in a tuple/list.
        """
        proc=par[0][1]
        evlist=par[0][2]
        proc.eventsFired=[]
        anyoccur=False
        for ev in evlist:
            if ev.occurred:
                anyoccur=True
                proc.eventsFired.append(ev)
                ev.occurred=False
        if anyoccur: #at least one event has fired; continue process
            _e._post(proc,at=_t,prior=1)

        else: #no event in list has fired, enter process in all ’waits’ lists
            proc.eventsFired=[]
            proc._nextTime=None #passivate calling process
            for ev in evlist:
                ev.queues.append([proc,evlist])

## begin waituntil functionality
def _test():
    """
    Gets called by simulate after every event, as long as there are processes
    waiting in condQ for a condition to be satisfied.
    Tests the conditions for all waiting processes. Where condition satisfied,
    reactivates that process immediately and removes it from queue.
    """
    global condQ
    rList=[]
    for el in condQ:
        if el.cond():
            rList.append(el)
            reactivate(el)
    for i in rList:
        condQ.remove(i)

    if not condQ:
        _stopWUStepping()

def _waitUntilFunc(proc,cond):
    global condQ
    """
    Puts a process ’proc’ waiting for a condition into a waiting queue.
    ’cond’ is a predicate function which returns True if the condition is
    satisfied.
    """
    if not cond():
        condQ.append(proc)
        proc.cond=cond
        _startWUStepping()         #signal ’simulate’ that a process is waiting
        # passivate calling process
        proc._nextTime=None
    else:
        #schedule continuation of calling process
        _e._post(proc,at=_t,prior=1)


##end waituntil functionality

def scheduler(till=0):
    """Schedules Processes/semi-coroutines until time ’till’.
    Deprecated since version 0.5.
    """
    simulate(until=till)

def holdfunc(a):
    a[0][1]._hold(a)

def requestfunc(a):
    """Handles ’yield request,self,res’ and ’yield (request,self,res),(<code>,self,par)’.
    <code> can be ’hold’ or ’waitevent’.
    """
    if type(a[0][0])==tuple:
        ## Compound yield request statement
        ## first tuple in ((request,self,res),(xx,self,yy))
        b=a[0][0]
        ## b[2]==res (the resource requested)
        ##process the first part of the compound yield statement
        ##a[1] is the Process instance
        b[2]._request(arg=(b,a[1]))
        ##deal with add-on condition to command
        ##Trigger processes for reneging
        class _Holder(Process):
            """Provides timeout process"""
            def trigger(self,delay):
                yield hold,self,delay
                if not proc in b[2].activeQ:
                    reactivate(proc)

        class _EventWait(Process):
            """Provides event waiting process"""
            def trigger(self,event):
                yield waitevent,self,event
                if not proc in b[2].activeQ:
                    a[1].eventsFired=self.eventsFired
                    reactivate(proc)

        #activate it
        proc=a[0][0][1] # the process to be woken up
        actCode=a[0][1][0]
        if actCode==hold:
            proc._holder=_Holder(name="RENEGE-hold for %s"%proc.name)
            ##                                          the timeout delay
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==waituntil:
            raise FatalSimerror("Illegal code for reneging: waituntil")
        elif actCode==waitevent:
            proc._holder=_EventWait(name="RENEGE-waitevent for %s"%proc.name)
            ##                                          the event
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==queueevent:
            raise FatalSimerror("Illegal code for reneging: queueevent")
        else:
            raise FatalSimerror("Illegal code for reneging %s"%actCode)
    else:
        ## Simple yield request command
        a[0][2]._request(a)

def releasefunc(a):
    a[0][2]._release(a)

def passivatefunc(a):
    a[0][1]._passivate(a)

```

```
def waitevfunc(a):
    #if waiting for one event only (not a tuple or list)
    evtpar=a[0][2]
    if isinstance(evtpar,SimEvent):
        a[0][2]._wait(a)
    # else, if waiting for an OR of events (list/tuple):
    else: #it should be a list/tuple of events
        # call _waitOR for first event
        evtpar[0]._waitOR(a)

def queueevfunc(a):
    #if queueing for one event only (not a tuple or list)
    evtpar=a[0][2]
    if isinstance(evtpar,SimEvent):
        a[0][2]._queue(a)
    #else, if queueing for an OR of events (list/tuple):
    else: #it should be a list/tuple of events
        # call _queueOR for first event
        evtpar[0]._queueOR(a)

def waituntilfunc(par):
    _waitUntilFunc(par[0][1],par[0][2])

def getfunc(a):
    """Handles ’yield get,self,buffer,what,priority’ and
    ’yield (get,self,buffer,what,priority),(<code>,self,par)’.
    <code> can be ’hold’ or ’waitevent’.
    """
    if type(a[0][0])==tuple:
        ## Compound yield request statement
        ## first tuple in ((request,self,res),(xx,self,yy))
        b=a[0][0]
        ## b[2]==res (the resource requested)
        ##process the first part of the compound yield statement
        ##a[1] is the Process instance
        b[2]._get(arg=(b,a[1]))
        ##deal with add-on condition to command
        ##Trigger processes for reneging
        class _Holder(Process):
            """Provides timeout process"""
            def trigger(self,delay):
                yield hold,self,delay
                #if not proc in b[2].activeQ:
                if proc in b[2].getQ:
                    reactivate(proc)

        class _EventWait(Process):
            """Provides event waiting process"""
            def trigger(self,event):
                yield waitevent,self,event
                if proc in b[2].getQ:
                    a[1].eventsFired=self.eventsFired
                    reactivate(proc)

        #activate it
        proc=a[0][0][1] # the process to be woken up
        actCode=a[0][1][0]
        if actCode==hold:
            proc._holder=_Holder("RENEGE-hold for %s"%proc.name)
            ##                                          the timeout delay
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==waituntil:
            raise FatalSimerror("Illegal code for reneging: waituntil")
        elif actCode==waitevent:
            proc._holder=_EventWait(proc.name)
            ##                                          the event
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==queueevent:
            raise FatalSimerror("Illegal code for reneging: queueevent")
        else:
            raise FatalSimerror("Illegal code for reneging %s"%actCode)
    else:
        ## Simple yield request command
        a[0][2]._get(a)


def putfunc(a):
    """Handles ’yield put’ (simple and compound hold/waitevent)
    """
    if type(a[0][0])==tuple:
        ## Compound yield request statement
        ## first tuple in ((request,self,res),(xx,self,yy))
        b=a[0][0]
        ## b[2]==res (the resource requested)
        ##process the first part of the compound yield statement
        ##a[1] is the Process instance
        b[2]._put(arg=(b,a[1]))
        ##deal with add-on condition to command
        ##Trigger processes for reneging
        class _Holder(Process):
            """Provides timeout process"""
            def trigger(self,delay):
                yield hold,self,delay
                #if not proc in b[2].activeQ:
                if proc in b[2].putQ:
                    reactivate(proc)

        class _EventWait(Process):
            """Provides event waiting process"""
            def trigger(self,event):
                yield waitevent,self,event
                if proc in b[2].putQ:
                    a[1].eventsFired=self.eventsFired
                    reactivate(proc)

        #activate it
        proc=a[0][0][1] # the process to be woken up
        actCode=a[0][1][0]
        if actCode==hold:
            proc._holder=_Holder("RENEGE-hold for %s"%proc.name)
            ##                                          the timeout delay
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==waituntil:
            raise FatalSimerror("Illegal code for reneging: waituntil")
        elif actCode==waitevent:
            proc._holder=_EventWait("RENEGE-waitevent for %s"%proc.name)
            ##                                          the event
            activate(proc._holder,proc._holder.trigger(a[0][1][2]))
        elif actCode==queueevent:
            raise FatalSimerror("Illegal code for reneging: queueevent")
        else:
            raise FatalSimerror("Illegal code for reneging %s"%actCode)
    else:
        ## Simple yield request command
        a[0][2]._put(a)

def simulate(until=0):
    """Schedules Processes/semi-coroutines until time 'until'"""

    """Gets called once. Afterwards, co-routines (generators) return by
    'yield' with a cargo:
    yield hold, self, <delay>: schedules the "self" process for activation
                               after <delay> time units.If <,delay> missing,
                               same as "yield hold,self,0"

    yield passivate,self    :  makes the "self" process wait to be re-activated

    yield request,self,<Resource>[,<priority>]: request 1 unit from <Resource>
        with <priority> pos integer (default=0)

    yield release,self,<Resource> : release 1 unit to <Resource>

    yield waitevent,self,<SimEvent>|[<Evt1>,<Evt2>,<Evt3), . . . ]:
        wait for one or more of several events


    yield queueevent,self,<SimEvent>|[<Evt1>,<Evt2>,<Evt3), . . . ]:
        queue for one or more of several events

    yield waituntil,self,cond : wait for arbitrary condition

    yield get,self,<buffer>[,<WhatToGet>[,<priority>]]
        get <WhatToGet> items from buffer (default=1);
        <WhatToGet> can be a pos integer or a filter function
        (Store only)

    yield put,self,<buffer>[,<WhatToPut>[,priority]]
        put <WhatToPut> items into buffer (default=1);
        <WhatToPut> can be a pos integer (Level) or a list of objects
        (Store)

    EXTENSIONS:
    Request with timeout reneging:
    yield (request,self,<Resource>),(hold,self,<patience>) :
        requests 1 unit from <Resource>. If unit not acquired in time period
        <patience>, self leaves waitQ (reneges).

    Request with event-based reneging:
    yield (request,self,<Resource>),(waitevent,self,<eventlist>):
        requests 1 unit from <Resource>. If one of the events in <eventlist> occurs before unit
        acquired, self leaves waitQ (reneges).

    Get with timeout reneging (for Store and Level):
    yield (get,self,<buffer>,nrToGet etc.),(hold,self,<patience>)
        requests <nrToGet> items/units from <buffer>. If not acquired <nrToGet> in time period
        <patience>, self leaves <buffer>.getQ (reneges).

    Get with event-based reneging (for Store and Level):
    yield (get,self,<buffer>,nrToGet etc.),(waitevent,self,<eventlist>)
        requests <nrToGet> items/units from <buffer>. If not acquired <nrToGet> before one of
        the events in <eventlist> occurs, self leaves <buffer>.getQ (reneges).



    Event notices get posted in event-list by scheduler after "yield" or by
    "activate"/"reactivate" functions.

    """
    global _endtime,_e,_stop,_t,_wustep
    _stop=False

    if _e is None:
        raise FatalSimerror("Simulation not initialized")
    if _e._isEmpty():
        message="SimPy: No activities scheduled"
        return message

    _endtime=until
    message="SimPy: Normal exit"
    dispatch={hold:holdfunc,request:requestfunc,release:releasefunc,
              passivate:passivatefunc,waitevent:waitevfunc,queueevent:queueevfunc,
              waituntil:waituntilfunc,get:getfunc,put:putfunc}
    commandcodes=dispatch.keys()
    commandwords={hold:"hold",request:"request",release:"release",passivate:"passivate",
        waitevent:"waitevent",queueevent:"queueevent",waituntil:"waituntil",
        get:"get",put:"put"}
    nextev=_e._nextev ## just a timesaver
    while not _stop and _t<=_endtime:
        try:
            a=nextev()
            if not a[0] is None:
                ## ’a’ is tuple "(<yield command>, <action>)"
                if type(a[0][0])==tuple:
                    ##allowing for yield (request,self,res),(waituntil,self,cond)
                    command=a[0][0][0]
                else:
                    command = a[0][0]
                if __debug__:
                    if not command in commandcodes:
                        raise FatalSimerror("Illegal command: yield %s"%command)
                dispatch[command](a)
        except FatalSimerror,error:
            print "SimPy: "+error.value
            sys.exit(1)
        except Simerror,error:
            message="SimPy: "+error.value
            _stop = True
        if _wustep:
            _test()
    _stopWUStepping()
    _e=None
    return message
```