Cheatsheet for SimPy version 2.0 (non-OO API)

Import statements

from SimPySimulation import *	Use SimPy simulation library
from SimPySimulationTrace import *	Use SimPy simulation library with tracing support
from SimPySimulationStep import *	Use SimPy simulation library with event-by-event execution support
from SimPySimulationRT import *	Use SimPy simulation library with real-time synchronization support

Basic program control and activate statements

initialize( )	Set the simulation clock to zero and initialize the run
simulate(until=endtime)	Start the simulation run; end it no later than *endtime* (NB: has additional parameters for SimulationStep or SimulationRT)
stopSimulation( )	Terminate the simulation immediately
activate(p,p.PEM(args),[{delay=0|at=now( )}, prior=False])	Activate entity *p* ; *delay* =activation delay; *at* =activation time; if *prior* == `True`, schedule *p* ahead of concurrently activated entities
reactivate(p,[{delay=0|at=now()}, prior=False])	Reactivate entity *p* ; *delay* =activation delay; *at* =activation time; if *prior* == `True`, schedule *p* ahead of concurrently activated entities
p.start(p.PEM(args),[{delay=0|at=now( )}, prior=False])	Activate entity *p* ; *delay* =activation delay; *at* =activation time; if *prior* ==True, schedule *p* ahead of concurrently activated entities. If the PEM is called ACTIONS and has no parameters, a shortcut form **p.start([{delay=0|at=now( )}, prior=False])** can be used.

Yield statements

yield hold,self,t	Suspend *self* 's PEM for a time delay of length *t*
yield passivate,self	Suspend *self* 's PEM until reactivated
yield waituntil,self,<condition>	Suspend *self* 's PEM until the <*condition*> becomes `True` (<*condition*> refers to name of a function that takes no parameters and returns a boolean indicating whether the state or condition has occurred)
yield waitevent,self,<events>	Suspend *self* 's PEM until some event in <*events*> occurs
yield queueevent,self,<events>	Suspend *self* 's PEM and insert it at the end of the queue of events awaiting the occurrence of some event in <*events*>
yield request,self,rR[,P]	Request a unit of *rR* with priority *P*
yield release,self,rR	Release a unit of *rR*
yield put,self,rL,q[,P]	Offer an amount *q* to Level *rL* with priority *P*
yield get,self,rL,q[,P]	Request an amount *q* from Level *rL* with priority *P*
yield put,self,rS,alist[,P]	Offer the list *alist* of items to Store *rS* with priority *P*
yield get,self,rS,which[,P]	If *which* is **integer**, request the first *which* items in Store *rS* with priority *P*. If *which* is a **filter-function** name, request the items selected by *which*

Yield statements with reneging clauses (compound yield)

yield (request,self,rR[,P]),(hold,self,t)	Request a unit of *rR* with priority *P* , but renege if time *t* passes before a unit is acquired
yield (request,self,rR[,P]), (waitevent,self,<events>)	Request a unit of *rR* with priority *P* , but renege if any event in <*events*> occurs before a resource unit is acquired
self.acquired(rR)	*(Obligatory after compound yield request.)* Return `True` if resource unit requested was acquired, `False` if self reneged

yield (put,self,rL,q[,P]), (hold,self,t)	Offer an amount *q* to Level *rL* with priority *P*, but renege if time *t* passes before there is room for *q* to be accepted
yield (put,self,rL,q[,P]), (waitevent,self,<events>)	Offer an amount *q* to Level *rL* with priority *P*, but renege if any event in <*events*> occurs before there is room for *q* to be accepted
yield (put,self,rS,alist[,P]),(hold,self,t)	Offer the list *alist* of items to Store *rS* with priority *P*, but renege if time *t* passes before there is space for them
yield (put,self,rS,alist[,P]),(waitevent,self,<events>)	Offer the list *alist* of items to Store *rS* with priority *P*, but renege if any event in <*events*> occurs before there is space for them
self.stored(rB)	*(Obligatory after compound yield put.)* Return `True` if amount or items were stored in *rB*, `False` if *self* reneged
yield (get,self,rL,q[,P]),(hold,self,t)	Request an amount *q* from Level *rL* with priority *P*, but renege if time *t* passes before amount *q* is acquired
yield (get,self,rL,q[,P]),(waitevent,self,<events>)	Request an amount *q* from Level *rL* with priority *P*, but renege if any event in <*events*> occurs before amount *q* is acquired
yield (get,self,rS,which[,P]),(hold,self,t)	If *which* is **integer**, request the first *which* items in Store *rS* with priority *P*. If *which* is a **filter-function name**, request the items selected by *which*, but renege if time *t* passes before they are acquired
yield (get,self,rS,which[,P]), (waitevent,self,<events>)	If *which* is **integer**, request the first *which* items in Store *rS* with priority P. If *which* is a **filter-function name**, request the items selected by *which*, but renege if any event in <*events*> occurs before they are acquired
self.acquired(rB)	*(Obligatory after compound yield get.)* Returns `True` if amount or items were acquired from rB, `False` if *self* reneged

Interrupt statements

self.cancel(p)	Delete all of process object *p*'s scheduled future actions
self.interrupt(pVictim)	Interrupt *pVictim* if it is active (*pVictim* cannot interrupt itself)
self.interrupted( )	Return `True` if *self*'s state is "interrupted"
self.interruptCause	Return the *p* that interrupted *self*
self.interruptLeft	Return the time to complete *pVictim*'s interrupted *yield hold*
self.interruptReset	Reset *self*'s state to “not interrupted”

SimEvent statements and attributes

SE = SimEvent(name='a_SimEvent')	Create the object *sE* of class SimEvent with the indicated property and the methods listed immediately below
sE.occurred	Return a boolean indicating whether *sE* has occurred
sE.waits	Return the list of *p*'s waiting for *sE*
sE.queues	Return the queue of *p*'s waiting for *sE*
sE.signal(None|<param>)	Cause *sE* to occur, and provide an optional "payload" <*param*> of any Python type
sE.signalparam	Return the payload <*param*> provided when *sE* last occurred
p.eventsFired	Return the list of events that were fired when *p* was last reactivated

Resource statements and attributes

rR = Resource(name='a_resource', unitName='a_unit', capacity=1, monitored={False|True}, monitorType={Monitor|Tally}, qType={FIFO|PriorityQ}, preemptable={False|True})	Create the object *rR* of class Resource with the indicated properties and the methods/properties listed immediately below where *qType* is *rR*'s *waitQ* discipline and the recorder objects exist only when `monitored==True`
rR.n	Return the number of *rR*'s units that are free
rR.waitQ	Return the queue of *p*'s waiting for one of *rR*'s units
rR.activeQ	Return the queue of *p*'s currently holding one of *rR*'s units
rR.waitMon	The recorder object observing *rR.waitQ*
rR.actMon	The recorder object observing *rR.actQ*

Level statements and attributes

rL = Level(name='a_level', unitName='a_unit', capacity='unbounded', monitored={False|True}, monitorType={Monitor|Tally}, initialBuffered={0|q}, putQType={FIFO|PriorityQ}, getQType={FIFO|PriorityQ})	Create the object *rL* of class Level with the indicated properties and the methods/properites listed immediately below where *'unbounded'* is interpreted as `sysmaxint`, *initialBuffered* is the initial amount of material in *rL* , and the recorder objects exist only when `monitored==True`
rL.amount	Return the amount of material in *rL*
rL.putQ	Return the queue of *p* 's waiting to add amounts to *rL*
rL.getQ	Return the queue of *p* 's waiting to get amounts from *rL*
rL.putQMon	The recorder object observing *rL.putQ*
rL.getQMon	The recorder object observing *rL.getQ*
rL.bufferMon	The recorder object observing *rL.amount*

Store statements and attributes

rS = Store(name='a_store', unitName='a_unit', capacity='unbounded', monitored={False|True}, monitorType={Monitor|Tally}, initialBuffered={None|<alist>}, putQType={FIFO|PriorityQ}, getQType={FIFO|PriorityQ})	Create the object *rS* of class Store with the indicated properties and the methods/properties listed immediately below where *'unbounded'* is interpreted as `sysmaxint`, *initialBuffered* is the initial (FIFO) queue of items in *rS* , and the recorder objects exist only when `monitored==True`
rS.theBuffer	Return the queue of items in *rS*
rS.nrBuffered	Return the number of items in *rS.theBuffer*
rS.putQ	Return the queue of *p* 's waiting to add items to *rS*
rS.getQ	Return the queue of *p* 's waiting to get items from *rS*
rS.putQMon	The recorder object observing *rS.putQ*
rS.getQMon	The recorder object observing *rS.getQ*
rS.bufferMon	The recorder object observing *rS.nrBuffered*

Monitor and Tally statements and attributes

rec = Monitor(name='a_Monitor', ylab='y', tlab='t')	Create the recorder object *rec* of class Monitor with the indicated properties and the methods listed immediately below
rec = Tally(name='a_Tally', ylab='y', tlab='t')	Create the recorder object *rec* of class Tally with the indicated properties and the methods listed immediately below
rec.observe(y,{now( )|t})	Record the value of *y* and the corresponding time, *now( )* or *t*
rec.reset({now( )|t})	Reset *rec* and initialize its starting time to *now( ) or t*
rec.count( )	Return *rec* 's current number of observations
rec.total( )	Return the sum of *rec* 's *y* -values
rec.mean( )	Return the sample average of *rec* 's *y* -values
rec.var( )	Return the sample variance of *rec* 's *y* -values
rec.timeAverage([now( )|t])	Return the time-duration-weighted average of *rec* 's *y* -values
rec.__str__( )	Return a string briefly describing *rec* 's current state
recMor[i]	Return *recMor* 's *i* -th observation as a sublist, $[t_i, y_i]$ (here and below, *recMor* is a recorder object of class Monitor)
recMor.yseries( )	Return *recMor* 's list of observed *y* -values, $[y_i]$
recMor.tseries( )	Return *recMor* 's list of observed *t* -values, $[t_i]$
recMor.histogram(low={0.0|mLo}, high={100.0|mHi}, nbins={10|mBi})	Return a histogram of *recMor* 's observations, using the indicated parameters
recTay.setHistogram(name=' ', low={0.0|tLo}, high={100.0|tHi}, nbins={10|tBi})	Create a histogram object to receive *recTay* 's updated counts (here and below, *recTay* is a recorder object of class Tally)
recTay.getHistogram( )	Return the histogram of *recTay* 's observations

SimulationTrace statements

trace.tchange({start=ts,}{end=te,} {toTrace=clist,}{outfile=fobj})	Change one or more trace parameters: *start* begins tracing at time *ts; end* stops tracing at time *te* ; *toTrace* limits the tracing to the yield commands given in the list of strings *clist* (default is ["hold","activate","cancel","reactivate","passivate","request", "release","interrupt","terminated","waitevent","queueevent", "signal","waituntil","put","get"]); *outfile* directs trace output to open, write-enabled file object *fobj* .
trace.treset( )	Resets tracing parameters to default
trace.tstart( )	Restarts tracing
trace.tstop( )	Stops tracing
trace.ttext(message)	Output string *message* just before next yield command trace output

SimPy identifiers (may not be overwritten)

FIFO, FatalSimerror, FireEvent, Histogram, JobEvt, JobEvtMulti, JobTO, Lister, Monitor, PriorityQ, Process, Queue, Resource, SimEvent, Simerror, Tally, trace,Trace, activate, allEventNotices, allEventTimes, askCancel, heapq, condQ, hold, holdfunc, initialize, now, passivate, passivatefunc, paused, queueevent, queueevfunc, reactivate, release, releasefunc, request, requestfunc, rtnow, rtstart, scheduler, simulate, simulateStep, startStepping, stopSimulation, stopStepping, sys, time, trace, types, waitevent, waitevfunc, waituntil, waituntilfunc, wallclock