



Cheatsheet for SimPy version 2.0 (non-OO API)

Import statements

<code>from SimPy.Simulation import *</code>	Use SimPy simulation library
<code>from SimPy.SimulationTrace import *</code>	Use SimPy simulation library with tracing
<code>from SimPy.SimulationStep import *</code>	Use SimPy simulation library with event-by-event execution
<code>from SimPy.SimulationRT import *</code>	Use SimPy simulation library with real-time synchronization
<code>from SimPy.SimulationGUIDebug import *</code>	Use SimPy simulation library with event-by-event GUI debugging

Basic program control and activate statements

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

Yield statements

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

Yield statements with renegeing clauses (compound yield)

<code>yield (request,self,rR,[P]),(hold,self,t)</code>	Request a unit of <i>rR</i> with priority <i>P</i> , but renege if time <i>t</i> passes before a unit is acquired
<code>yield (request,self,rR,[P]), (waitevent,self,<events>)</code>	Request a unit of <i>rR</i> with priority <i>P</i> , but renege if any event in <i><events></i> occurs before a resource unit is acquired
<code>self.acquired(rR)</code>	(Obligatory after compound yield request.) Return <i>True</i> if resource unit requested was acquired, <i>False</i> if self renegeed
<code>yield (put,self,rL,q,[P]), (hold,self,t)</code>	Offer an amount <i>q</i> to Level <i>rL</i> with priority <i>P</i> , but renege if time <i>t</i> passes before there is room for <i>q</i> to be accepted

<code>yield (put,self,rL,q[,P]), (waitevent,self,<events>)</code>	Offer an amount q to Level rL with priority P , but renege if any event in $\langle events \rangle$ occurs before there is room for q to be accepted
<code>yield (put,self,rS,alist[,P]),(hold,self,t)</code>	Offer the list $alist$ of items to Store rS with priority P , but renege if time t passes before there is space for them
<code>yield (put,self,rS,alist[,P]),(waitevent,self,<events>)</code>	Offer the list $alist$ of items to Store rS with priority P , but renege if any event in $\langle events \rangle$ occurs before there is space for them
<code>self.stored(rB)</code>	(Obligatory after compound yield put.) Return <i>True</i> if amount or items were stored in rB , <i>False</i> if <i>self</i> reneged
<code>yield (get,self,rL,q[,P]),(hold,self,t)</code>	Request an amount q from Level rL with priority P , but renege if time t passes before amount q is acquired
<code>yield (get,self,rL,q[,P]),(waitevent,self,<events>)</code>	Request an amount q from Level rL with priority P , but renege if any event in $\langle events \rangle$ occurs before amount q is acquired
<code>yield (get,self,rS,which[,P]),(hold,self,t)</code>	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
<code>yield (get,self,rS,which[,P]), (waitevent,self,<events>)</code>	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 $\langle events \rangle$ occurs before they are acquired
<code>self.acquired(rB)</code>	(Obligatory after compound yield get.) Returns <i>True</i> if amount or items were acquired from rB , <i>False</i> if <i>self</i> reneged

Interrupt statements

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

SimEvent statements and attributes

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

Resource statements and attributes

<code>rR = Resource(name='a_resource', unitName='a_unit', capacity=1, monitored={False True}, monitorType={Monitor Tally}, qType={FIFO PriorityQ}, preemptable={False True})</code>	Create the object rR of class <i>Resource</i> with the indicated properties and the methods/properties listed immediately below where $qType$ is rR 's <i>waitQ</i> discipline and the recorder objects exist only when <code>monitored==True</code>
<code>rR.n</code>	Return the number of rR 's units that are free
<code>rR.waitQ</code>	Return the queue of p 's waiting for one of rR 's units
<code>rR.activeQ</code>	Return the queue of p 's currently holding one of rR 's units
<code>rR.waitMon</code>	The recorder object observing $rR.waitQ$
<code>rR.actMon</code>	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 <i>rL</i> of class Level with the indicated properties and the methods/properties listed immediately below where 'unbounded' is interpreted as <code>sys.maxint</code> , <i>initialBuffered</i> is the initial amount of material in <i>rL</i> , and the recorder objects exist only when <code>monitored==True</code>
rL.amount	Return the amount of material in <i>rL</i>
rL.putQ	Return the queue of <i>p</i> 's waiting to add amounts to <i>rL</i>
rL.getQ	Return the queue of <i>p</i> 's waiting to get amounts from <i>rL</i>
rL.putQMon	The recorder object observing <i>rL.putQ</i>
rL.getQMon	The recorder object observing <i>rL.getQ</i>
rL.bufferMon	The recorder object observing <i>rL.amount</i>

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 <i>rS</i> of class Store with the indicated properties and the methods/properties listed immediately below where 'unbounded' is interpreted as <code>sys.maxint</code> , <i>initialBuffered</i> is the initial (FIFO) queue of items in <i>rS</i> , and the recorder objects exist only when <code>monitored==True</code>
rS.theBuffer	Return the queue of items in <i>rS</i>
rS.nrBuffered	Return the number of items in <i>rS.theBuffer</i>
rS.putQ	Return the queue of <i>p</i> 's waiting to add items to <i>rS</i>
rS.getQ	Return the queue of <i>p</i> 's waiting to get items from <i>rS</i>
rS.putQMon	The recorder object observing <i>rS.putQ</i>
rS.getQMon	The recorder object observing <i>rS.getQ</i>
rS.bufferMon	The recorder object observing <i>rS.nrBuffered</i>

Monitor and Tally statements and attributes

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

SimulationTrace statements

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

SimulationGUIDEbug statements

register (<i>obj</i> , <i>hook</i> , <i>name</i>)	Registers an object to create windows in GUI. <i>obj</i> : any object or SimPy.Process subclass. <i>hook</i> : a function that returns a string to print user-defined text in window. <i>name</i> : a string to be used as the window title
--	---

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, rnow, rtstart, scheduler, simulate, simulateStep, startStepping, stopSimulation, stopStepping, sys, time, trace, types, waitevent, waitevfunc, waituntil, waituntilfunc, wallclock
--