



# Cheatsheet for SimPy version 1.9

## Import statements

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

## 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 <i>ACTIONS</i> 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,&lt;condition&gt;</code>	Suspend <i>self</i> 's PEM until the <i>&lt;condition&gt;</i> becomes True ( <i>&lt;condition&gt;</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,&lt;events&gt;</code>	Suspend <i>self</i> 's PEM until some event in <i>&lt;events&gt;</i> occurs
<code>yield queueevent,self,&lt;events&gt;</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>&lt;events&gt;</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 <b>integer</b> , request the first <i>which</i> items in Store <i>rS</i> with priority <i>P</i> . If <i>which</i> is a <b>filter-function</b> 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,&lt;events&gt;)</code>	Request a unit of <i>rR</i> with priority <i>P</i> , but renege if any event in <i>&lt;events&gt;</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

## SimPy 1.9 Cheatsheet

<code>yield (put,self,rL,q[,P]), (waitevent,self,&lt;events&gt;)</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,&lt;events&gt;)</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 <code>True</code> if amount or items were stored in $rB$ , <code>False</code> if $self$ renege
<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,&lt;events&gt;)</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 <b>integer</b> , request the first $which$ items in Store $rS$ with priority $P$ . If $which$ is a <b>filter-function name</b> , 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,&lt;events&gt;)</code>	If $which$ is <b>integer</b> , request the first $which$ items in Store $rS$ with priority $P$ . If $which$ is a <b>filter-function name</b> , 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 <code>True</code> if amount or items were acquired from $rB$ , <code>False</code> if $self$ renege

### 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 <code>True</code> if $self$ 's state is "interrupted"
<code>self.interruptCause</code>	Return the $p$ that interrupted $self$
<code>self.interruptLeft</code>	Return the time to complete $pVictim$ 's interrupted <i>yield hold</i>
<code>self.interruptReset</code>	Reset $self$ 's state to "not interrupted"

### SimEvent statements and attributes

<code>SE = SimEvent(name='a_SimEvent')</code>	Create the object $sE$ of class <code>SimEvent</code> 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.queues</code>	Return the queue of $p$ 's waiting for $sE$
<code>sE.signal(None&lt;param&gt;)</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 <code>Resource</code> 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

<b>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})</b>	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>sysmaxint</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>
<b>rL.amount</b>	Return the amount of material in <i>rL</i>
<b>rL.putQ</b>	Return the queue of <i>p</i> 's waiting to add amounts to <i>rL</i>
<b>rL.getQ</b>	Return the queue of <i>p</i> 's waiting to get amounts from <i>rL</i>
<b>rL.putQMon</b>	The recorder object observing <i>rL.putQ</i>
<b>rL.getQMon</b>	The recorder object observing <i>rL.getQ</i>
<b>rL.bufferMon</b>	The recorder object observing <i>rL.amount</i>

### Store statements and attributes

<b>rS = Store(name='a_store', unitName='a_unit', capacity='unbounded', monitored={False True}, monitorType={Monitor Tally}, initialBuffered={None &lt;alist&gt;}, putQType={FIFO PriorityQ}, getQType={FIFO PriorityQ})</b>	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>sysmaxint</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>
<b>rS.theBuffer</b>	Return the queue of items in <i>rS</i>
<b>rS.nrBuffered</b>	Return the number of items in <i>rS.theBuffer</i>
<b>rS.putQ</b>	Return the queue of <i>p</i> 's waiting to add items to <i>rS</i>
<b>rS.getQ</b>	Return the queue of <i>p</i> 's waiting to get items from <i>rS</i>
<b>rS.putQMon</b>	The recorder object observing <i>rS.putQ</i>
<b>rS.getQMon</b>	The recorder object observing <i>rS.getQ</i>
<b>rS.bufferMon</b>	The recorder object observing <i>rS.nrBuffered</i>

### Monitor and Tally statements and attributes

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

## SimulationTrace statements

<code>trace.tchange({start=ts},{end=te,} {toTrace=clist},{outfile=fobj})</code>	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> .
<code>trace.treset()</code>	Resets tracing parameters to default
<code>trace.tstart()</code>	Restarts tracing
<code>trace.tstop()</code>	Stops tracing
<code>trace.ttext(message)</code>	Output string <i>message</i> 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, queueefunc, reactivate, release, releasefunc, request, requestfunc, rnow, rtstart, scheduler, simulate, simulateStep, startStepping, stopSimulation, stopStepping, sys, time, trace, types, waitevent, waitefunc, waituntil, waituntilfunc, wallclock