SimPy Documentation

Release 2.3.3

Klaus Müller, Tony Vignaux, Ontje Lünsdorf, Stefan Scherfke

Feb 24, 2018

Contents

SimPy is a process-based discrete-event simulation language based on standard Python. It provides the modeller with components of a simulation model including *processes*, for active components like customers, messages, and vehicles, and *resources*, for passive components that form limited capacity congestion points like servers, checkout counters, and tunnels. It also provides *monitor variables* to aid in gathering statistics. Random variates are provided by the standard Python *random* module.

It is based on ideas from Simula and Simscript and provides efficient implementation of co-routines using Python's generators capability. It requires Python 2.7 or later including Python 3.x. It was first released in 2002 under the GNU LGPL.

Contents:

CHAPTER 1

Getting Started

1.1 Installation

This file describes the installation of SimPy 2.3.3.

1. Check that you have Python 2.7 or above.
2. You can install SimPy easily via PIP

   ```
   $ pip install SimPy
   ```

   If SimPy is already installed, use the *-U* option for pip to upgrade:

   ```
   $ pip install -U SimPy
   ```

   Remember, on Linux/MacOS/Unix you may need *root* privileges to install SimPy. This also applies to the installing SimPy manually, as described below.

3. To manually install a SimPy tarball, or to execute the examples, download and unpack the SimPy archive into a folder (using option "Use folder names" in WinZip, "Re-create folders" in Linux Archive Manager, or similar option in your unpacker). This will create a SimPy-2.3.3 folder with all source code and documentation.

   Open a terminal, *cd* to the SimPy folder and execute *setup.py* or *pip install .*:

   ```
   $ cd where/you/put/simpy/SimPy-x.y
   $ python setup.py install
   $ # or
   $ pip install .
   ```

   If you do not have permissions to perform the installation as root, you can install SimPy into a non-standard folder:

   ```
   $ cd where/you/put/simpy/SimPy-x.y
   $ python setup.py install --home <dir>
   ```

4. Run one or more of the programs under *docs/examples* to see whether Python finds the SimPy module. If you get an error message like *ImportError: No module named SimPy*, move the SimPy folder into a directory which you know to be on the Python module search path (like /Lib/site-packages).
5. The tutorial and manuals are in the *docs/html* folder. Many users have commented that the Bank tutorials are valuable in getting users started on building their own simple models. Even a few lines of Python and SimPy can model significant real systems.

For more help, contact the SimPy-Users mailing list. SimPy users are pretty helpful.

Enjoy simulation programming in SimPy!

1.2 Contents of This SimPy Distribution

SimPy 2.3.3 contains the following files:

- SimPy - Python code directory for the SimPy 2.3.3 package
  - Lister.py, a prettyprinter for class instances
  - Simulation.py, code for SimPy simulation
  - SimulationTrace.py, code for simulation with tracing
  - SimulationStep.py, code for executing simulations event-by-event
  - SimulationRT.py, code for synchronizing simulation time with wallclock time
  - SimulationGUIDebug.py, code for debugging/event stepping of models with a GUI
  - SimGUI.py, code for generating a Tk-based GUI for SimPy simulations
  - SimPlot.py, code for generating Tk-based plots (screen and Postscript)
  - __init__.py, initialisation of SimPy package
- Tests - a directory containing tests for simpy
- **docs - a directory containing the complete, browseable (HTML) documentation of SimPy.** It includes tutorials and descriptions of accessing external packages from SimPy. **Click on index.html!**
- docs/examples - some SimPy models (in traditional and Object Oriented API)
  - Bankmodels - a sub-directory with the models of the Bank tutorials (in traditional and Object Oriented API)
- LICENSE.txt - GNU Lesser General Public Licence text

1.3 Changes from Release 2.3.3

This section addresses the difference between previous SimPy version 2.1.0 and 2.3.3 in terms of changes and additions.

1.3.1 Changes from 2.1.0 to 2.2.b1

- The Unit tests have been rewritten
- The directory sturcture of the release has been simplified
- The documentation has had some minor changes

1.4 COMPATIBILITY: SimPy

SimPy has been used successfully with many packages and modules, such as Tk/Tkinter for GUIs and VPython and matplotlib for graphical output.

The design of SimPy is such that no incompatibilities with Python 2.7 through 3.6 modules or Python 2.7 through 3.6-accessible packages are expected.

SimPy 2.3.3 has been tested with Python 2.7, 3.4, 3.5, 3.6. On Linux and Windows 10.

Should SimPy users discover any incompatibilities, the authors would be grateful for a report. Just send a message with the problem and its context to: simpy-users@lists.sourceforge.net.

1.5 SimPy History

SimPy is based on ideas from Simula and Simscript but uses standard Python. It combines two previous packages, SiPy, in Simula-Style (Klaus Muller) and SimPy, in Simscript style (Tony Vignaux and Chang Chui)

SimPy is based on efficient implementation of co-routines using Python's generators capability.

The package has been hosted on Sourceforge.net since 15 September 2002. Sourceforge.net's service has always been outstanding. It is essential to the SimPy project! Thanks, all you people at SourceForge!

1.5.1 December 2011: Release 2.3

- Support for Python 3.x has been added
- Examples and tutorials modified to run on Python 2.6 and up including Python 3.
- Examples can now be executed via py.test so we can make sure they do run.
- The documentation has had some reorganisation. The index has had work done on it. The Simple manual has been pulled out and is setup as a separate manual.

1.5.2 September 2011: Release 2.2b1

- The Unit tests have been rewritten.
- The directory sturcture of the release has been simplified
- The documentation has had some minor changes

1.5.3 May 2010: Version 2.1.0

A major release of SimPy, with a new code base, a (small) number of additions to the API, and added documentation.

Additions

- A function *step* has been added to the API. When called, it executes the next scheduled event. (*step* is actually a method of Simulation.)
- Another new function is *peek*. It returns the time of the next event. By using *peek* and *step* together, one can easily write e.g. an interactive program to step through a simulation event by event.

- A simple interactive debugger `stepping.py` has been added. It allows stepping through a simulation, with options to skip to a certain time, skip to the next event of a given process, or viewing the event list.
- Versions of the Bank tutorials (documents and programs) using the advanced object-oriented API have been added.
- A new document describes tools for gaining insight into and debugging SimPy models.

Changes

- Major re-structuring of SimPy code, resulting in much less SimPy code – great for the maintainers.
- Checks have been added which test whether entities belong to the same *Simulation* instance.
- The *Monitor* and *Tally* methods *timeAverage* and *timeVariance* now calculate only with the observed time-series. No value is assumed for the period prior to the first observation.
- Changed class *Lister* so that circular references between objects no longer lead to stack overflow and crash.

Repairs

- Functions *allEventNotices* and *allEventTimes* are working again.
- Error messages for methods in SimPy.Lib work again.

1.5.4 April 2009: Release 2.0.1

A bug-fix release of SimPy 2.0

1.5.5 October 2008: Version 2.0

This is a major release with changes to the SimPy application programming interface (API) and the formatting of the documentation.

API changes

In addition to its existing API, SimPy now also has an object oriented API. The additional API

- allows running SimPy in parallel on multiple processors or multi-core CPUs,
- supports better structuring of SimPy programs,
- allows subclassing of class *Simulation* and thus provides users with the capability of creating new simulation modes/libraries like SimulationTrace, and
- reduces the total amount of SimPy code, thereby making it easier to maintain.

Note that the OO API is **in addition** to the old API. SimPy 2.0 is fully backward compatible.

Documentation format changes

SimPy's documentation has been restructured and processed by the Sphinx documentation generation tool. This has generated one coherent, well structured document which can be easily browsed. A seach capability is included.

1.5.6 March 2008: Version 1.9.1

This is a bug-fix release which cures the following bugs:

- Excessive production of circular garbage, due to a circular reference between Process instances and event notices. This led to large memory requirements.
- Runtime error for preempts of proceeses holding multiple Resource objects.

It also adds a Short Manual, describing only the basic facilities of SimPy.

1.5.7 December 2007: Version 1.9

This is a major release with added functionality/new user API calls and bug fixes.

Major changes

- The event list handling has been changed to improve the runtime performance of large SimPy models (models with thousands of processes). The use of dictionaries for timestamps has been stopped. Thanks are due to Prof. Norm Matloff and a team of his students who did a study on improving SimPy performance. This was one of their recommendations. Thanks, Norm and guys! Furthermore, in version 1.9 the 'heapq' sorting package replaces 'bisect'. Finally, cancelling events no longer removes them, but rather marks them. When their event time comes, they are ignored. This was Tony Vignaux' idea!
- The Manual has been edited and given an easier-to-read layout.
- The Bank2 tutorial has been extended by models which use more advanced SimPy commands/constructs.

Bug fixes

- The tracing of 'activate' statements has been enabled.

Additions

- A method returning the time-weighted variance of observations has been added to classes Monitor and Tally.
- A shortcut activation method called "start" has been added to class Process.

1.5.8 January 2007: Version 1.8

Major Changes

- SimPy 1.8 and future releases will not run under the obsolete Python 2.2 version. They require Python 2.3 or later.
- The Manual has been thoroughly edited, restructured and rewritten. It is now also provided in PDF format.
- The Cheatsheet has been totally rewritten in a tabular format. It is provided in both XLS (MS Excel spreadsheet) and PDF format.
- The version of SimPy.Simulation(RT/Trace/Step) is now accessible by the variable 'version'.
- The *__str__* method of Histogram was changed to return a table format.

Bug fixes

- Repaired a bug in *yield waituntil* runtime code.
- Introduced check for *capacity* parameter of a Level or a Store being a number > 0.
- Added code so that self.eventsFired gets set correctly after an event fires in a compound yield get/put with a waitevent clause (reneging case).
- Repaired a bug in prettyprinting of Store objects.

Additions

- New compound yield statements support time-out or event-based reneging in get and put operations on Store and Level instances.
- *yield get* on a Store instance can now have a filter function.
- All Monitor and Tally instances are automatically registered in list *allMonitors* and *allTallies*, respectively.
- The new function *startCollection* allows activation of Monitors and Tallies at a specified time.
- A *printHistogram* method was added to Tally and Monitor which generates a table-form histogram.
- In SimPy.SimulationRT: A function for allowing changing the ratio wall clock time to simulation time has been added.

1.5.9 June 2006: Version 1.7.1

This is a maintenance release. The API has not been changed/added to.

- Repair of a bug in the _get methods of Store and Level which could lead to synchronization problems (blocking of producer processes, despite space being available in the buffer).
- Repair of Level __init__ method to allow initialBuffered to be of either float or int type.
- Addition of type test for Level get parameter 'nrToGet' to limit it to positive int or float.
- To improve pretty-printed output of 'Level' objects, changed attribute '_nrBuffered' to 'nrBuffered' (synonym for 'amount' property).
- To improve pretty-printed output of 'Store' objects, added attribute 'buffered' (which refers to '_theBuffer' attribute).

1.5.10 February 2006: Version 1.7

This is a major release.

- Addition of an abstract class Buffer, with two sub-classes *Store* and *Level* Buffers are used for modelling inter-process synchronization in producer/ consumer and multi-process cooperation scenarios.
- Addition of two new *yield* statements:
  - *yield put* for putting items into a buffer, and
  - *yield get* for getting items from a buffer.
- The Manual has undergone a major re-write/edit.

- All scripts have been restructured for compatibility with IronPython 1 beta2. This was doen by moving all *import* statements to the beginning of the scripts. After the removal of the first (shebang) line, all scripts (with the exception of plotting and GUI scripts) can run successfully under this new Python implementation.

1.5.11 September 2005: Version 1.6.1

This is a minor release.

- Addition of Tally data collection class as alternative to Monitor. It is intended for collecting very large data sets more efficiently in storage space and time than Monitor.
- Change of Resource to work with Tally (new Resource API is backwards-compatible with 1.6).
- Addition of function setHistogram to class Monitor for initializing histograms.
- New function allEventNotices() for debugging/teaching purposes. It returns a prettyprinted string with event times and names of process instances.
- Addition of function allEventTimes (returns event times of all scheduled events).

1.5.12 15 June 2005: Version 1.6

- Addition of two compound yield statement forms to support the modelling of processes reneging from resource queues.
- Addition of two test/demo files showing the use of the new reneging statements.
- Addition of test for prior simulation initialization in method activate().
- Repair of bug in monitoring thw waitQ of a resource when preemption occurs.
- Major restructuring/editing to Manual and Cheatsheet.

1.5.13 1 February 2005: Version 1.5.1

- MAJOR LICENSE CHANGE:

  Starting with this version 1.5.1, SimPy is being release under the GNU Lesser General Public License (LGPL), instead of the GNU GPL. This change has been made to encourage commercial firms to use SimPy in for-profit work.

- Minor re-release
- No additional/changed functionality
- Includes unit test file'MonitorTest.py' which had been accidentally deleted from 1.5
- Provides updated version of 'Bank.html' tutorial.
- Provides an additional tutorial ('Bank2.html') which shows how to use the new synchronization constructs introduced in SimPy 1.5.
- More logical, cleaner version numbering in files.

1.5.14 1 December 2004: Version 1.5

- No new functionality/API changes relative to 1.5 alpha
- Repaired bug related to waiting/queuing for multiple events
- SimulationRT: Improved synchronization with wallclock time on Unix/Linux

1.5.15 25 September 2004: Version 1.5alpha

- New functionality/API additions
  - SimEvents and signalling synchronization constructs, with 'yield waitevent' and 'yield queueevent' commands.
  - A general "wait until" synchronization construct, with the 'yield waituntil' command.
- No changes to 1.4.x API, i.e., existing code will work as before.

1.5.16 19 May 2004: Version 1.4.2

- Sub-release to repair two bugs:
  - The unittest for monitored Resource queues does not fail anymore.
  - SimulationTrace now works correctly with "yield hold,self" form.
- No functional or API changes

1.5.17 29 February 2004: Version 1.4.1

- Sub-release to repair two bugs:
  - The (optional) monitoring of the activeQ in Resource now works correctly.
  - The "cellphone.py" example is now implemented correctly.
- No functional or API changes

1.5.18 1 February 2004: Version 1.4 published on SourceForge

1.5.19 22 December 2003: Version 1.4 alpha

- New functionality/API changes
  - All classes in the SimPy API are now new style classes, i.e., they inherit from *object* either directly or indirectly.
  - Module *Monitor.py* has been merged into module *Simulation.py* and all *SimulationXXX.py* modules. Import of *Simulation* or any *SimulationXXX* module now also imports *Monitor*.
  - Some *Monitor* methods/attributes have changed. See Manual!
  - *Monitor* now inherits from *list*.
  - A class *Histogram* has been added to *Simulation.py* and all *SimulationXXX.py* modules.
  - A module *SimulationRT* has been added which allows synchronization between simulated and wallclock time.

  - A moduleSimulationStep which allows the execution of a simulation model event-by-event, with the facility to execute application code after each event.
  - A Tk/Tkinter-based module *SimGUI* has been added which provides a SimPy GUI framework.
  - A Tk/Tkinter-based module *SimPlot* has been added which provides for plot output from SimPy programs.

1.5.20 22 June 2003: Version 1.3

- No functional or API changes
- Reduction of sourcecode linelength in Simulation.py to <= 80 characters

1.5.21 9 June 2003: Version 1.3 alpha

- Significantly improved performance
- Significant increase in number of quasi-parallel processes SimPy can handle
- New functionality/API changes:
  - Addition of SimulationTrace, an event trace utility
  - Addition of Lister, a prettyprinter for instance attributes
  - No API changes
- Internal changes:
  - Implementation of a proposal by Simon Frost: storing the keys of the event set dictionary in a binary search tree using bisect. Thank you, Simon! SimPy 1.3 is dedicated to you!
- Update of Manual to address tracing.
- Update of Interfacing doc to address output visualization using Scientific Python gplt package.

1.5.22 29 April 2003: Version 1.2

- No changes in API.
- **Internal changes:**
  - Defined "True" and "False" in Simulation.py to support Python 2.2.

1.5.23 22 October 2002:

- Re-release of 0.5 Beta on SourceForge.net to replace corrupted file __init__.py.
- No code changes whatever!

1.5.24 18 October 2002:

- Version 0.5 Beta-release, intended to get testing by application developers and system integrators in preparation of first full (production) release. Released on SourceForge.net on 20 October 2002.
- More models

- Documentation enhanced by a manual, a tutorial ("The Bank") and installation instructions.
- Major changes to the API:
  - Introduced 'simulate(until=0)' instead of 'scheduler(till=0)'. Left 'scheduler()' in for backward compatibility, but marked as deprecated.
  - Added attribute "name" to class Process. Process constructor is now:

```
def __init__(self,name="a_process")
```

    Backward compatible if keyword parameters used.

  - Changed Resource constructor to:

```
def __init__(self,capacity=1,name="a_resource",unitName="units")
```

    Backward compatible if keyword parameters used.

1.5.25 27 September 2002:

- Version 0.2 Alpha-release, intended to attract feedback from users
- Extended list of models
- Upodated documentation

1.5.26 17 September 2002

- Version 0.1.2 published on SourceForge; fully working, pre-alpha code
- Implements simulation, shared resources with queuing (FIFO), and monitors for data gathering/analysis.
- Contains basic documentation (cheatsheet) and simulation models for test and demonstration.

1.6 SimPy Resources

SimPy can be downloaded from the "SimPy web-site": https://github.com/SimPyClassic/SimPyClassic

Simulation model developers are encouraged to share their SimPy modeling techniques with the SimPy community.

Software developers are encouraged to interface SimPy with other Python-accessible packages, such as GUI, data base or mapping and to share these new capabilities with the community under the GNU GPL.

Feature requests for future SimPy versions should be sent to "Klaus G. Muller", kgmuller at users.sourceforge.net, or "Tony Vignaux", vignaux at users.sourceforge.net.

CHAPTER 2

Manuals

2.1 SimPy Classic Manual

Authors

- Tony Vignaux <Vignaux@users.sourceforge.net>
- Klaus Muller <Muller@users.sourceforge.net>
- Bob Helmbold

Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7 and later

Date Feb 24, 2018

Contents

This document describes SimPy Classic version 2.3.3. Changes from the previous version are listed in *Appendix A0*.

Note: This document does **not** describe the object oriented (OO) API which has been added to SimPy with version 2.0. SimPy 2.0 is fully backward compatible with previous versions. The procedural API and the OO API co-exist happily in SimPy 2.x.

2.1.1 Introduction

SimPy is a Python-based discrete-event simulation system that models active components such as messages, customers, trucks, planes by parallel processes. It provides a number of tools for the simulation programmer including *Processes* to model active entities, three kinds of resource facilities (*Resources*, *Levels*, and *Stores*) and ways of recording results by using *Monitors* and *Tallys*.

The basic active elements of a SimPy model are process objects (i.e., objects of a Process class – see *Processes*). As a general practice and for brevity we will often refer to both process objects and their classes as "processes." Thus, "process" may refer to a Process class or to a process object, depending on context. To avoid ambiguity or for added emphasis we often explicitly state whether a class or an object is intended. In addition we will use "entity" to refer to process objects as this is frequently used in the simulation literature. Here, though, we restrict it to process objects and it will not be used for any other elements in the simulation.

During the simulation, Process objects may be delayed for fixed or random times, queued at resource facilities, and may be interrupted by or interact in other ways with other processes and components. For example, Automobiles in a model of a gas station may have to queue while waiting for a pump to become available . Once obtaining a pump it takes some time to fill before releasing the pump.

A SimPy script contains the declaration of one or more Process classes and the creation of process objects (entities) from them. Each process object executes its *Process Execution Method* (referred to later as a *PEM*), a method that determines its actions. Each PEM runs in parallel with (and may interact with) the PEMs of other process objects.

There are three types of resource facilities (*Resources*, *Levels*, and *Stores*). Each type models a congestion point where process objects may have to queue while waiting to acquire or, in some cases to deposit, a resource.

Resources have several *resource units*, each of which may be used by process objects. Extending the example above, the gas station might be modelled as a resource with its pumps as resource units. On receiving a request for a pump from a car, the gas station resource automatically queues waiting cars until one becomes available. The pump resource unit is held by the car until it is released for possible use by another car.

Levels model the supply and consumption of a homogeneous undifferentiated "material." The Level at any time holds an amount of material that is fully described by a scalar (real or integer). This can be increased or decreased by process objects. For example, a gas (petrol) station stores gas in large storage tanks. The tanks can be increased by Tanker deliveries and reduced by cars refuelling. A car need not return the gas to the Level in contrast to the requirement for Resource units.

Stores model the production and consumption of individual items. A store hold a list of items. Process objects can insert or remove items from the list. For example, surgical procedures (treated as process objects) require specific lists of personnel and equipment that may be treated as the items in a Store facility such as a clinic or hospital. The items held in a Store can be of any Python type. In particular they can be process objects, and this may be exploited to facilitate modelling Master/Slave relationships.

In addition to the number of free units or quantities, resource facilities all hold queues of waiting process objects which are operated automatically by SimPy. They also operate a reneging mechanism so that a process object can abandon the wait.

Monitors and *Tallys* are used to compile statistics as a function of time on variables such as waiting times and queue lengths. These statistics consist of simple averages and variances, time-weighted averages, or histograms. They can be gathered on the queues associated with Resources, Levels and Stores. For example we may collect data on the average number of cars waiting at a gas station and the distribution of their waiting times. Tallys update the current statistics as the simulation progresses, but cannot preserve complete time-series records. Monitors can preserve complete time-series records that may later be used for more advanced post-simulation analyses.

Before attempting to use SimPy, you should be able to write Python code. In particular, you should be able to define and use classes and their objects. Python is free and usable on most platforms. We do not expound it here. You can find out more about it and download it from the Python web-site (https://www.python.org). SimPy requires *Python* 2.3 or later.

[Return to *Top*]

2.1.2 Simulation with SimPy

To use the SimPy simulation system you must import its `Simulation` module (or one of the *alternatives*):

```
from SimPy.Simulation import *
```

All discrete-event simulation programs automatically maintain the current simulation time in a software clock. This cannot be changed by the user directly. In SimPy the current clock value is returned by the `now()` function.

At the start of the simulation the software clock is set to 0.0. While the simulation program runs, simulation time steps forward from one *event* to the next. An event occurs whenever the state of the simulated system changes. For example, an event might be the arrival or departure of a car from the gas station.

The following statement initializes global simulation variables and sets the software clock to zero. It must appear in the script before any SimPy process objects are activated.

```
initialize( )
```

This is followed by SimPy statements creating and activating process objects. Activation of process objects adds events to the simulation schedule. Execution of the simulation itself starts with the following statement:

`simulate(until=`*endtime*`)`

The simulation starts, and SimPy seeks and executes the first scheduled event. Having executed that event, the simulation seeks and executes the next event, and so on.

Typically a simulation is terminated when *endtime* is reached but it can be stopped at any time by the command:

```
stopSimulation( )
```

`now( )` will then equal the time when this was called. The simulation will also stop if there are no more events to execute (so `now()` equals the time the last scheduled event occurred)

After the simulation has stopped, further statements can be executed. `now()` will retain the time of stopping and data held in Monitors will be available for display or further analysis.

The following fragment shows only the *main* block in a simulation program. (Complete, runnable examples are shown in *Example 1* and *Example 2*). Here `Message` is a (previously defined) Process class and `m` is defined as an object of that class, that is, a particular message. Activating `m` has the effect of scheduling at least one event by starting the PEM of `m` (here called `go`). The `simulate(until=1000.0)` statement starts the simulation itself, which immediately jumps to the first scheduled event. It will continue until it runs out of events to execute or the simulation time reaches `1000.0`. When the simulation stops the (previously written) `Report` function is called to display the results:

```
initialize()
m = Message()
activate(m, m.go(), at=0.0)
simulate(until=1000.0)

Report()  #  report results when the simulation finishes
```

The object-oriented interface

An object-oriented API interface was added in SimPy 2.0. It is described more fully in `SimPyOO_API`. It defines a class of *Simulation* objects and makes running multiple simulations cleaner and easier. It is compatible with the procedural version described in this Manual. Using the object-oriented API, the program fragment listed at the end of the previous subsection would look like this:

```
s = Simulation()
s.initialize()
m = Message(sim=s)
s.activate(m, m.go(), at=0.0)
s.simulate(until=1000.0)

Report()  #  report results when the simulation finishes
```

Further examples of the OO style exist in the *SimPyModels* directory and the *Bank Tutorial*.

Alternative SimPy simulation libraries

In addition to *SimPy.Simulation*, SimPy provides four alternative simulation libraries which have the basic `SimPy.Simulation` capabilities, plus additional facilities:

- *SimPy.SimulationTrace* for program tracing: With `from SimPy.SimulationTrace import *`, any SimPy program automatically generates detailed event-by-event tracing output. This makes the library ideal for program development/testing and for teaching SimPy.
- *SimPy.SimulationRT* for real time synchronization: `from SimPy.SimulationRT import *` facilitates synchronizing simulation time and real (wall-clock) time. This capability can be used to implement, e.g., interactive game applications or to demonstrate a model's execution in real time.
- *SimPy.SimulationStep* for event-stepping through a simulation: The import `from SimPy.SimulationStep import *` provides an API for stepping through a simulation event by event. This can assist with debugging models, interacting with them on an event-by-event basis, getting event-by-event output from a model (e.g. for plotting purposes), etc.
- *SimPy.SimulationGUIDebug* for event-stepping through a simulation with a GUI: `from SimPy.SimulationGUIDebug import *` provides an API for stepping through a simulation event-by-event, with a GUI for user control. The event list, Process and Resource objects are shown in windows. This is useful for debugging models and for teaching discrete event simulation with SimPy.

[Return to *Top*]

2.1.3 Processes

The active objects for discrete-event simulation in SimPy are process objects – instances of some class that inherits from SimPy's Process class.

For example, if we are simulating a computing network we might model each message as an object of the class `Message`. When message objects arrive at the computing network they make transitions between nodes, wait for service at each one, are served for some time, and eventually leave the system. The `Message` class specifies all the actions of each message in its Process Execution Method (PEM). Individual message objects are created as the simulation runs, and their evolutions are directed by the `Message` class's PEM.

Defining a process

Each Process class inherits from SimPy's `Process` class. For example the header of the definition of a new `Message` Process class would be:

```
class Message(Process):
```

At least one Process Execution Method (PEM) must be defined in each Process class[1]. A PEM may have arguments in addition to the required `self` argument that all methods must have. Naturally, other methods and, in particular, an `__init__` method, may be defined.

- `A Process Execution Method (PEM)` defines the actions that are performed by its process objects. Each PEM must contain at least one of the `yield` statements, described later. This makes it a Python generator function so that it has resumable execution – it can be restarted again after the yield statement without losing its current state. A PEM may have any name of your choice. For example it may be called `execute( )` or `run( )`.

  "The `yield` statements are simulation commands which affect an ongoing life-cycle of Process objects. These statements control the execution and synchronization of multiple processes. They can delay a process, put it to sleep, request a shared resource or provide a resource. They can add new events on the simulation event schedule, cancel existing ones, or cause processes to wait for a state change."

  For example, here is a the Process Execution Method, `go(self)`, for the `Message` class. Upon activation it prints out the current time, the message object's identification number and the word "Starting". After a simulated delay of 100.0 time units (in the `yield hold, ...` statement) it announces that this message object has "Arrived":

```
def go(self):
    print(now(), self.i, 'Starting')
    yield hold, self, 100.0
    print(now(), self.i, 'Arrived')
```

  A process object's PEM starts execution when the object is activated, provided the `simulate(until= ...)` statement has been executed.

- **__init__(self, ...)**, where ... indicates method arguments. This method initializes the process object, setting values for some or all of its attributes. As for any sub-class in Python, the first line of this method must call the `Process` class's `__init__( )` method in the form:

```
Process.__init__(self)
```

  You can then use additional commands to initialize attributes of the Process class's objects. You can also override the standard `name` attribute of the object.

  The `__init__( )` method is always called whenever you create a new process object. If you do not wish to provide for any attributes other than a `name`, the `__init__` method may be dispensed with. An example of an `__init__( )` method is shown in the example below.

[1] The variable `version`, imported from `SimPy.Simulation`, contains the revision number and date of the current version.

Creating a process object

An entity (process object) is created in the usual Python manner by calling the class. Process classes have a single argument, `name` which can be specified if no `__init__` method is defined. It defaults to `'a_process'`. It can be over-ridden if an `__init__` method is defined.

For example to create a new `Message` object with a name `Message23`:

```
m = Message(name="Message23")
```

Note: When working through this and all other SimPy manuals, the reader is encouraged to type in, run and experiment with all examples as she goes. No better way of learning exists than **doing**! A suggestion: if you want to see how a SimPy model is being executed, *trace* it by replacing *from SimPy.Simulation import ** with *from SimPy.SimulationTrace import **. Any Python environment is suitable – an interactive Python session, IDLE, IPython, Scite ...

Example 1: This is is a complete, runnable, SimPy script. We declare a `Message` class and define an `__init__( )` method and a PEM called `go( )`. The `__init__( )` method provide an instance variables of an identification number and message length. We do not actually use the `len` attribute in this example.

Two `messages`, `p1` and `p2` are created. `p1` and `p2` are activated to start at simulation times 0.0 and 6.0, respectively. Nothing happens until the `simulate(until=200)` statement. When both messages have finished (at time 6.0+100.0=106.0) there will be no more events so the simulation will stop at that time:

```
from SimPy.Simulation import Process, activate, initialize, hold, now,␣
↪simulate


class Message(Process):
    """A simple Process"""

    def __init__(self, i, len):
        Process.__init__(self, name='Message' + str(i))
        self.i = i
        self.len = len

    def go(self):
        print('%s %s %s' % (now(), self.i, 'Starting'))
        yield hold, self, 100.0
        print('%s %s %s' % (now(), self.i, 'Arrived'))


initialize()
p1 = Message(1, 203)    # new message
activate(p1, p1.go())   # activate it
p2 = Message(2, 33)
activate(p2, p2.go(), at=6.0)
simulate(until=200)
print('Current time is %s' % now())   # will print 106.0
```

Running this program gives the following output:

```
0 1 Starting
6.0 2 Starting
```

```
100.0 1 Arrived
106.0 2 Arrived
Current time is 106.0
```

Elapsing time in a Process

A *PEM* uses the `yield hold` command to temporarily delay a process object's operations.

yield hold

`yield hold, self,`*t*

Causes the process object to delay *t* time units[2]. After the delay, it continues with the next statement in its PEM. During the `hold` the object's operations are suspended.

Example 2: In this example the Process Execution Method, `buy`, has an extra argument, `budget`:

```
from SimPy.Simulation import Process, activate, hold, initialize, simulate


class Customer(Process):
    def buy(self, budget=0):
        print('Here I am at the shops %s' % self.name)
        t = 5.0
        for i in range(4):
            yield hold, self, t
            # executed 4 times at intervals of t time units
            print('I just bought something %s' % self.name)
            budget -= 10.00
        print('All I have left is %s I am going home %s' % (budget, self.
↪name))


initialize()

# create a customer named "Evelyn",
C = Customer(name='Evelyn')

# and activate her with a budget of 100
activate(C, C.buy(budget=100), at=10.0)

simulate(until=100.0)
```

Starting and stopping SimPy Process Objects

A process object is "passive" when first created, i.e., it has no scheduled events. It must be *activated* to start its Process Execution Method. To activate an instance of a Process class you can use either the `activate` function or the `start` method of the Process. (see the *Glossary* for an explanation of the modified Backus-Naur Form (BNF) notation used).

[2] More than one can be defined but only one can be executed by any process object.

activate

- `activate(`*p, p.pemname([args])*`[,{at=now()|delay=0}][,prior=False])`

  activates process object *p*, provides its Process Execution Method *p.pemname( )* with arguments *args* and possibly assigns values to the other optional parameters. The default is to activate at the current time (`at=now( )`) with no delay (`delay=0.0`) and `prior` set to `False`. You may assign other values to `at`, `delay`, and `prior`.

  Example: to activate a process object, `cust` with name `cust001` at time 10.0 using a PEM called `lifetime`:

```
activate(cust, cust.lifetime(name='cust001'), at=10.0)
```

  However, `delay` overrides `at`, in the sense that when a `delay=`*period* clause is included, then activation occurs at `now( )` or `now( )`+*period* (whichever is larger), irrespective of what value of *t* is assigned in the `at=`*t* clause. This is true even when the value of *period* in the delay clause is zero, or even negative. So it is better and clearer to choose one (or neither) of `at=`*t* and `delay=`*period*, but not both.

  If you set `prior=True`, then process object *p* will be activated *before* any others that happen to be scheduled for activation at the same time. So, if several process objects are scheduled for activation at the same time and all have `prior=True`, then the last one scheduled will actually be the first to be activated, the next-to-last of those scheduled, the second to be activated, and so forth.

  Retroactive activations that attempt to activate a process object before the current simulation time terminate the simulation with an error report.

start

An alternative to `activate()` function is the `start` method. There are a number of ways of using it:

- *p.*`start(`*p.pemname([args])*`[,{at=now()|delay=0}][,prior=False])`

  is an alternative to the `activate` statement. *p* is a Process object. The generator function, *pemname*, can have any identifier (such as `run`, `life-cycle`, etc). It can have parameters.

  For example, to activate the process object `cust` using the PEM with identifier, `lifetime` at time 10.0 we would use:

```
cust.start(cust.lifetime(name='cust001'), at=10.0)
```

- *p.*`start([`*p.*`ACTIONS()] [,{at=now()|delay=0}][,prior=False])`

  if `p` is a Process object and the generator function is given the *standard identifier*, `ACTIONS`. `ACTIONS`, is recognized as a Process Execution Method. It may *not* have parameters. The call *p.*`ACTIONS()` is optional.

  For example, to activate the process object `cust` with the standard PEM identifier `ACTIONS` at time 10.0, the following are equivalent (and the second version is more convenient):

```
cust.start(cust.ACTIONS(), at=10.0)
cust.start(at=10.0)
```

- An *anonymous* instance of Process class *PR* can be created and activated in one command using `start` with the standard PEM identifier, `ACTIONS`.

  PR.`([args]).start( [,{at=now()|delay=0}][,prior=False])`

  Here, *PR* is the identifier for the Process class and not for a Process object as was *p*, in the statements above. The generator method `ACTIONS` may *not* have parameters.

For example, if `Customer` is a SimPy Process class we can create and activate an anonymous instance at time 10.0:

```
Customer(name='cust001').start(at=10.0)
```

You can use the `passivate`, `reactivate`, or `cancel` commands to control Process objects.

passivate

- `yield passivate, self`

  suspends the process object itself. It becomes "passive". To get it going again another process must `reactivate` it.

reactivate

- `reactivate(`*p*`[,{at=now()|delay=0}][,prior=False])`

  reactivates a passive process object, *p*. It becomes "active". The optional parameters work as for `activate`. A process object cannot reactivate itself. To temporarily suspend itself it must use `yield hold,self,t` instead.

cancel

- `self.cancel(`*p*`)`

  deletes all scheduled future events for process object *p*. A process cannot `cancel` itself. If that is required, use `yield passivate,self` instead. Only "active" process objects can be canceled.

A process object is "terminated" after all statements in its process execution method have been completed. If the object is still referenced by a variable, it becomes just a data container. This can be useful for extracting information. Otherwise, it is automatically destroyed.

Even activated process objects will not start operating until the `simulate(until=`*endtime*`)` statement is executed. This starts the simulation going and it will continue until time *endtime* (unless it runs out of events to execute or the command `stopSimulation( )` is executed).

Example 3 This simulates a firework with a time fuse. We have put in a few extra `yield hold` commands for added suspense.

```
from SimPy.Simulation import Process, activate, initialize, hold, now, simulate


class Firework(Process):

    def execute(self):
        print('%s firework launched' % now())
        yield hold, self, 10.0  # wait 10.0 time units
        for i in range(10):
            yield hold, self, 1.0
            print('%s tick' % now())
        yield hold, self, 10.0  # wait another 10.0 time units
        print('%s Boom!!' % now())
```

```
initialize()
f = Firework()  # create a Firework object, and
# activate it (with some default parameters)
activate(f, f.execute(), at=0.0)
simulate(until=100)
```

Here is the output. No formatting was attempted so it looks a bit ragged:

```
0 firework launched
11.0 tick
12.0 tick
13.0 tick
14.0 tick
15.0 tick
16.0 tick
17.0 tick
18.0 tick
19.0 tick
20.0 tick
30.0 Boom!!
```

A source fragment

One useful program pattern is the *source*. This is a process object with a Process Execution Method (PEM) that sequentially generates and activates other process objects – it is a source of other process objects. Random arrivals can be modelled using random intervals between activations.

Example 4: A source. Here a source creates and activates a series of customers who arrive at regular intervals of 10.0 units of time. This continues until the simulation time exceeds the specified `finishTime` of 33.0. (Of course, to model customers with random inter-arrival times the `yield hold` statement would use a random variate, such as `expovariate( )`, instead of the constant 10.0 inter-arrival time used here.) The following example assumes that the `Customer` class has previously been defined with a PEM called `run` that does not require any arguments:

```
class Source(Process):

    def execute(self, finish):
        while now() < finish:
            c = Customer()          # create a new customer object, and
            # activate it (using default parameters)
            activate(c, c.run())
            print('%s %s) % (now(), 'customer')
            yield hold, self, 10.0

initialize()
g = Source()                        # create the Source object, g,
                                    # and activate it
activate(g, g.execute(finish=33.0), at=0.0)
simulate(until=100)
```

Asynchronous interruptions

An active process object can be interrupted by another but cannot interrupt itself.

interrupt

- `self.interrupt(victim)`

  The *interrupter* process object uses its `interrupt` method to interrupt the *victim* process object. The interrupt is just a *signal*. After this statement, the *interrupter* process object continues its PEM.

  For the interrupt to have an immediate effect, the *victim* process object must be *active* – that is it must have an event scheduled for it (that is, it is "executing" a `yield hold`). If the *victim* is not active (that is, it is either *passive* or *terminated*) the interrupt has no effect. For example, process objects queuing for resource facilities cannot be interrupted because they are *passive* during their queuing phase.

If interrupted, the *victim* returns from its `yield hold` statement prematurely. It must then check to see if it has been interrupted by calling:

interrupted

- `self.interrupted( )`

  which returns `True` if it has been interrupted. The *victim* can then either continue in the current activity or switch to an alternative, making sure it tidies up the current state, such as releasing any resources it owns.

interruptCause

- `self.interruptCause`

  when the *victim* has been interrupted, `self.interruptCause` is a reference to the *interrupter* object.

interruptLeft

- `self.interruptLeft`

  gives the time remaining in the interrupted `yield hold`. The interruption is reset (that is, "turned off") at the *victim's* next call to a `yield hold`.

..index:: interruptReset

interruptReset

- `self.interruptReset( )`

  will reset the interruption.

It may be helpful to think of an interruption signal as instructing the *victim* to determine whether it should interrupt itself. If the *victim* determines that it should interrupt itself, it then becomes responsible for making any necessary readjustments – not only to itself but also to any other simulation components that are affected. (The *victim* must take responsibility for these adjustments, because it is the only simulation component that "knows" such details as whether or not it is interrupting itself, when, and why.)

Example 5. A simulation with interrupts. A bus is subject to breakdowns that are modelled as interrupts caused by a `Breakdown` process. Notice that the `yield hold,self,tripleft` statement may be interrupted, so if the `self.interrupted()` test returns `True` a reaction to it is required. Here, in addition to delaying the bus for repairs, the reaction includes scheduling the next breakdown. In this example the `Bus` Process class does not require an `__init__()` method:

```
from SimPy.Simulation import (Process, activate, hold, initialize, now,
                              reactivate, simulate)


class Bus(Process):

    def operate(self, repairduration, triplength):  # PEM
        tripleft = triplength                    # time needed to finish trip
        while tripleft > 0:
            yield hold, self, tripleft          # try to finish the trip
            if self.interrupted():              # if another breakdown occurs
                print('%s at %s' % (self.interruptCause.name, now()))
                tripleft = self.interruptLeft    # time to finish the trip
                self.interruptReset()            # end interrupt state
                reactivate(br, delay=repairduration)  # restart breakdown br
                yield hold, self, repairduration      # delay for repairs
                print('Bus repaired at %s' % now())
            else:
                break  # no more breakdowns, bus finished trip
        print('Bus has arrived at %s' % now())


class Breakdown(Process):
    def __init__(self, myBus):
        Process.__init__(self, name='Breakdown ' + myBus.name)
        self.bus = myBus

    def breakBus(self, interval):       # process execution method
        while True:
            yield hold, self, interval  # breakdown interarrivals
            if self.bus.terminated():
                break
            self.interrupt(self.bus)    # breakdown to myBus


initialize()
b = Bus('Bus')       # create a bus object
activate(b, b.operate(repairduration=20, triplength=1000))
br = Breakdown(b)    # create breakdown br to bus b
activate(br, br.breakBus(300))
simulate(until=4000)
print('SimPy: No more events at time %s' % now())
```

The output from this example:

```
Breakdown Bus at 300
Bus repaired at 320
Breakdown Bus at 620
Bus repaired at 640
Breakdown Bus at 940
Bus repaired at 960
Bus has arrived at 1060
```

```
SimPy: No more events at time 1260
```

The bus finishes at 1060 but the simulation finished at 1260. Why? The `breakdowns` PEM consists of a loop, one breakdown following another at `300` intervals. The last breakdown finishes at 960 and then a breakdown event is scheduled for 1260. But the bus finished at 1060 and is not affected by the breakdown. These details can easily be checked by importing from `SimPy.SimulationTrace` and re-running the program.

Where interrupts can occur, the victim of interrupts must test for interrupt occurrence after every appropriate `yield hold` and react appropriately to it. A victim holding a resource facility when it gets interrupted continues to hold it.

Advanced synchronization/scheduling capabilities

The preceding scheduling constructs all depend on specified time values. That is, they delay processes for a specific time, or use given time parameters when reactivating them. For a wide range of applications this is all that is needed.

However, some applications either require or can profit from an ability to activate processes that must wait for other processes to complete. For example, models of real-time systems or operating systems often use this kind of approach. *Event* Signalling is particularly helpful in such situations. Furthermore, some applications need to activate processes when certain conditions occur, even though when (or if) they will occur may be unknown. SimPy has a general *wait until* to support clean implementation of this approach.

This section describes how SimPy provides *event* Signalling and *wait until* capabilities.

Creating and Signalling SimEvents

As mentioned in the Introduction, for ease of expression when no confusion can arise we often refer to both process objects and their classes as "processes", and mention their object or class status only for added clarity or emphasis. Analogously, we will refer to objects of SimPy's `SimEvent` class as "SimEvents"[3] (or, if no confusion can arise, simply as "events"). However, we sometimes mention their object or class character for clarity or emphasis.

SimEvent objects must be created before they can be fired by a `signal`. You create the SimEvent object, `sE`, from SimPy's `SimEvent` class by a statement like the following:

```
sE = SimEvent(name='I just had a great new idea!')
```

A SimEvent's `name` attribute defaults to `a_SimEvent` unless you provide your own, as shown here. Its `occurred` attribute, `sE.occurred`, is a Boolean that defaults to `False`. It indicates whether the event `sE` has occurred.

You program a SimEvent to "occur" or "fire" by "signalling" it like this:

`sE.signal(`*<payload parameter>*`)`

This "signal" is "received" by all processes that are either "waiting" or "queueing" for this event to occur. What happens when they receive this signal is explained in the next section. The *<payload parameter>* is optional – it defaults to `None`. It can be of any Python type. Any process can retrieve it from the event's `signalparam` attribute, for example by:

```
message = sE.signalparam
```

[3] unless it is further delayed by being *interrupted*.
This is used to model any elapsed time an entity might be involved in. For example while it is passively being provided with service.

Waiting or Queueing for SimEvents

You can program a process either to "wait" or to "queue" for the occurrence of SimEvents. The difference is that *all* processes "waiting" for some event are reactivated as soon as it occurs. For example, all firemen go into action when the alarm sounds. In contrast, only the *first* process in the "queue" for some event is reactivated when it occurs. That is, the "queue" is FIFO[5]. An example might be royal succession – when the present ruler dies: "The king is dead. Long live the (new) king!" (And all others in the line of succession move up one step.)

You program a process to `wait` for SimEvents by including in its PEM:

yield waitevent

- `yield waitevent, self,`<*events part*>

  where <*events part*> can be either:

  - one SimEvent object, e.g. `myEvent`, or
  - a tuple of SimEvent objects, e.g. `(myEvent,myOtherEvent,TimeOut)`, or
  - a list of SimEvent objects, e.g. `[myEvent,myOtherEvent,TimeOut]`

  If none of the events in the <*events part*> have occurred, the process is passivated and joined to the list of processes waiting for some event in <*events part*> to occur (or to recur).

  On the other hand, when *any* of the events in the <*events part*> occur, then *all* of the processes "waiting" for those particular events are reactivated at the current time. Then the `occurred` flag of those particular events is reset to `False`. Resetting their `occurred` flag prevents the waiting processes from being constantly reactivated. (For instance, we do not want firemen to keep responding to any such "false alarms.") For example, suppose the <*events part*> lists events `a, b` and `c` in that order. If events `a` and `c` occur, then all of the processes waiting for event `a` are reactivated. So are all processes waiting for event `c` but not `a`. Then the `occurred` flags of events `a` and `c` are toggled to `False`. No direct changes are made to event `b` or to any processes waiting for it to occur.

You program a process to "queue" for events by including in its PEM:

yield queueevent

- `yield queueevent, self,`<*events part*>

  where the <*events part*> is as described above.

  If none of the events in the <*events part*> has occurred, the process is passivated and appended to the FIFO queue of processes queuing for some event in <*events part*> to occur (or recur).

  But when any of the events in <*events part*> occur, the process at the head of the "queue" is taken off the queue and reactivated at the current time. Then the `occurred` flag of those events that occurred is reset to `False` as in the "waiting" case.

Finding Which Processes Are Waiting/Queueing for an Event, and Which Events Fired

SimPy automatically keeps current lists of what processes are "waiting" or "queueing" for SimEvents. They are kept in the `waits` and `queues` attributes of the SimEvent object and can be read by commands like the following:

[5] "First-in-First-Out" or FCFS, "First-Come-First-Served"

```
TheProcessesWaitingFor_myEvent = myEvent.waits
TheProcessesQueuedFor_myEvent = myEvent.queues
```

However, you should not attempt to change these attributes yourself.

Whenever `myEvent` occurs, i.e., whenever a `myEvent.signal(...)` statement is executed, SimPy does the following:

- If there are any processes waiting or queued for that event, it reactivates them as described in the preceding section.
- If there are no processes waiting or queued (i.e., `myEvent.waits` and `myEvent.queues` are both empty), it toggles `myEvent.occurred` to `True`.

SimPy also automatically keeps track of which events were fired when a process object was reactivated. For example, you can get a list of the events that were fired when the object `Godzilla` was reactivated with a statement like this:

```
GodzillaRevivedBy = Godzilla.eventsFired
```

Example 6. This complete SimPy script illustrates these constructs. (It also illustrates that a Process class may have more than one PEM. Here the `Wait_Or_Queue` class has two PEMs – `waitup` and `queueup`.):

```
from SimPy.Simulation import *


class Wait_Or_Queue(Process):
    def waitup(self, myEvent):      # PEM illustrating "waitevent"
                                    # wait for "myEvent" to occur
        yield waitevent, self, myEvent
        print('At %s, some SimEvent(s) occurred that activated object %s.' %
              (now(), self.name))
        print('   The activating event(s) were %s' %
              ([x.name for x in self.eventsFired]))

    def queueup(self, myEvent):     # PEM illustrating "queueevent"
                                    # queue up for "myEvent" to occur
        yield queueevent, self, myEvent
        print('At %s, some SimEvent(s) occurred that activated object %s.' %
              (now(), self.name))
        print('   The activating event(s) were %s' %
              ([x.name for x in self.eventsFired]))


class Signaller(Process):
    # here we just schedule some events to fire
    def sendSignals(self):
        yield hold, self, 2
        event1.signal()         # fire "event1" at time 2
        yield hold, self, 8
        event2.signal()         # fire "event2" at time 10
        yield hold, self, 5
        event1.signal()         # fire all four events at time 15
        event2.signal()
        event3.signal()
        event4.signal()
        yield hold, self, 5
        event4.signal()         # event4 recurs at time 20
```

```
initialize()

# Now create each SimEvent and give it a name
event1 = SimEvent('Event-1')
event2 = SimEvent('Event-2')
event3 = SimEvent('Event-3')
event4 = SimEvent('Event-4')
Event_list = [event3, event4]   # define an event list

s = Signaller()
# Activate Signaller "s" *after* events created
activate(s, s.sendSignals())

w0 = Wait_Or_Queue('W-0')
# create object named "W-0", and set it to
# "waitup" for SimEvent "event1" to occur
activate(w0, w0.waitup(event1))
w1 = Wait_Or_Queue('W-1')
activate(w1, w1.waitup(event2))
w2 = Wait_Or_Queue('W-2')
activate(w2, w2.waitup(Event_list))
q1 = Wait_Or_Queue('Q-1')
# create object named "Q-1", and put it to be first
# in the queue for Event_list to occur
activate(q1, q1.queueup(Event_list))
q2 = Wait_Or_Queue('Q-2')
# create object named "Q-2", and append it to
# the queue for Event_list to occur
activate(q2, q2.queueup(Event_list))

simulate(until=50)
```

This program outputs:

```
At 2, some SimEvent(s) occurred that activated object W-0.
   The activating event(s) were ['Event-1']
At 10, some SimEvent(s) occurred that activated object W-1.
   The activating event(s) were ['Event-2']
At 15, some SimEvent(s) occurred that activated object W-2.
   The activating event(s) were ['Event-3']
At 15, some SimEvent(s) occurred that activated object Q-1.
   The activating event(s) were ['Event-3', 'Event-4']
At 20, some SimEvent(s) occurred that activated object Q-2.
   The activating event(s) were ['Event-4']
```

Each output line, `The activating event(s) were ...`, lists the contents of the named object's `eventsFired` attribute. One of those events "caused" the object to reactivate at the indicated time. Note that at time 15 objects `W-0` and `W-1` were not affected by the recurrence of `event1` and `event2` because they already were active. Also at time 15, even though objects `W-2`, `Q-1` and `Q-2` were all waiting for `event3`, only `W-2` and `Q-1` were reactivated. Process object `Q-2` was not reactivated at that time because it was not first in the queue. Finally, `Q-2` was reactivated at time 20, when `event4` fired again.

"waituntil" synchronization – waiting for any condition

SimPy provides the `waituntil` feature that makes a process's progress depend on the state of the simulation. This is useful if, for example, you need to reactivate a process when (if ever) the simulation enters the state `goodWeather OR (nrCustomers>50 AND price<22.50)`. Doing that requires *interrogative* scheduling, while all other SimPy synchronization constructs are *imperative* – i.e., the condition must be tested after every change in state until it becomes `True`.

This requires that after every change in system state SimPy must run a special (hidden) process that tests and responds appropriately to the condition's truth-value. This clearly takes more run time than SimPy's imperative scheduling constructs. So SimPy activates its interrogative testing process only so long as at least one process is executing a `waituntil` statement. When this is not the case, the run time overhead is minimal (about 1 percent extra run time).

yield waituntil

You program a process to wait for a condition to be satisfied by including in its PEM a statement of the form:

```
yield waituntil, self,<cond>
```

where <*cond*> is a reference to a function, without parameters, that returns a Boolean value indicating whether the simulation state or condition to be waited for has occurred.

Example 7. This program using the `yield waituntil ...` statement. Here the function `killed()`, in the `life()` PEM of the `Player` process, defines the condition to be waited for:

```
from SimPy.Simulation import (Process, initialize, activate, simulate,
                              hold, now, waituntil, stopSimulation)
import random


class Player(Process):

    def __init__(self, lives=1, name='ImaTarget'):
        Process.__init__(self, name)
        self.lives = lives
        # provide Player objects with a "damage" property
        self.damage = 0

    def life(self):
        self.message = 'Drat! Some %s survived Federation attack!' % \
            (target.name)

        def killed():     # function testing for "damage > 5"
            return self.damage > 5

        while True:
            yield waituntil, self, killed
            self.lives -= 1
            self.damage = 0
            if self.lives == 0:
                self.message = '%s wiped out by Federation at \
                time %s!' % (target.name, now())
                stopSimulation()


class Federation(Process):
```

```
    def fight(self):                # simulate Federation operations
        print('Three %s attempting to escape!' % (target.name))
        while True:
            if random.randint(0, 10) < 2:  # check for hit on player
                target.damage += 1         # hit! increment damage to player
                if target.damage <= 5:     # target survives
                    print('Ha! %s hit! Damage= %i' %
                          (target.name, target.damage))
                else:
                    if (target.lives - 1) == 0:
                        print('No more %s left!' % (target.name))
                    else:
                        print('Now only %i %s left!' % (target.lives - 1,
                                                        target.name))

            yield hold, self, 1


initialize()
gameOver = 100
# create a Player object named "Romulans"
target = Player(lives=3, name='Romulans')
activate(target, target.life())
# create a Federation object
shooter = Federation()
activate(shooter, shooter.fight())
simulate(until=gameOver)
print(target.message)
```

One possible output from this program is shown below. Whether the Romulans are wiped out or some escape depends on what simulation states the randomization feature produces:

```
Three Romulans attempting to escape!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Ha! Romulans hit! Damage= 3
Ha! Romulans hit! Damage= 4
Ha! Romulans hit! Damage= 5
Now only 2 Romulans left!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Ha! Romulans hit! Damage= 3
Ha! Romulans hit! Damage= 4
Ha! Romulans hit! Damage= 5
Now only 1 Romulans left!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Drat! Some Romulans survived Federation attack!
```

The `waituntil` construct is so general that in principle it could replace all the other synchronization approaches (but at a run time cost).

[Return to *Top*]

2.1.4 Resources

The three resource facilities provided by SimPy are *Resources*, *Levels* and *Stores*. Each models a congestion point where process objects may have to queue up to obtain resources. This section describes the Resource type of resource facility.

An example of queueing for a Resource might be a manufacturing plant in which a `Task` (modelled as a *process object*) needs work done by a `Machine` (modelled as a Resource object). If all of the `Machines` are currently being used, the `Task` must wait until one becomes free. A SimPy Resource can have a number of identical `units`, such as a number of identical `machine` units. A process obtains a unit of the Resource by `requesting` it and, when it is finished, `releasing` it. A Resource maintains a list of process objects that have requested but not yet received one of the Resource's units (called the `waitQ`), and another list of processes that are currently using a unit (the `activeQ`). SimPy creates and updates these queues itself – the user can access them, but should not change them.

Defining a Resource object

A Resource object, `r`, is established by the following statement:

```
r = Resource(capacity=1, name='a_resource', unitName='units',
             qType=FIFO, preemptable=False,
             monitored=False, monitorType=Monitor)
```

where

- `capacity` is a positive real or integer value that specifies the total number of identical units in Resource object `r`.
- `name` is a descriptive name for this Resource object (e.g., `'gasStation'`).
- `unitName` is a descriptive name for a unit of the resource (e.g., `'pump'`).
- `qType` is either `FIFO`[5] or `PriorityQ`. It specifies the queue discipline of the resource's `waitQ`; typically, this is `FIFO` and that is the default value. If `PriorityQ` is specified, then higher-priority requests waiting for a unit of Resource `r` are inserted into the `waitQ` ahead of lower priority requests. See *Priority requests for a Resource unit* for details.
- `preemptable` is a Boolean (`False` or `True`); typically, this is `False` and that is the default value. If it is `True`, then a process requesting a unit of this resource may preempt a lower-priority process in the `activeQ`, i.e., one that is already using a unit of the resource. See *Preemptive requests for a Resource unit* for details.
- `monitored` is a boolean (`False` or `True`). If set to `True`, then information is gathered on the sizes of `r`'s `waitQ` and `activeQ`, otherwise not.
- `monitorType` is either `Monitor` or `Tally` and indicates the type of *Recorder* to be used (see *Recording Resource queue lengths* for an example and additional discussion).

Each Resource object, `r`, has the following additional attributes:

- `r.n`, the number of units that are currently free.
- `r.waitQ`, a queue (list) of processes that have requested but not yet received a unit of `r`, so `len(r.waitQ)` is the number of process objects currently waiting.
- `r.activeQ`, a queue (list) of process objects currently using one of the Resource's units, so `len(r.activeQ)` is the number of units that are currently in use.
- `r.waitMon`, the record (made by a `Monitor` or a `Tally` whenever `monitored==True`) of the activity in `r.waitQ`. So, for example, `r.waitMon.timeaverage()` is the average number of processes in `r.waitQ`. See *Recording Resource queue lengths* for an example.

- `r.actMon`, the record (made by a `Monitor` or a `Tally` whenever `monitored==True`) of the activity in `r.activeQ`.

Requesting and releasing a unit of a Resource

A process can request and later release a unit of the Resource object, `r`, by using the following yield commands in a Process Execution Method:

yield request

- `yield request,self,`*r [,P=0]*

  requests a unit of Resource *r* with (optional) real or integer priority value *P*. If no priority is specified, it defaults to 0. Larger values of *P* represent higher priorities. See the following sections on *Queue Order* for more information on how priority values are used. Although this form of request can be used for either `FIFO` or `PriorityQ` priority types, these values are *ignored* when `qType==FIFO`.

yield release

```
yield release, self, r
```

releases the unit of *r*.

Queue Order

If a requesting process must wait it is placed into the resource's `waitQ` in an order determined by settings of the resource's `qType` and `preemptable` attributes and of the priority value it uses in the `request` call.

Non-priority queueing

If the `qType` is not specified it takes the presumed value of `FIFO`[5]. In that case processes wait in the usual first-come-first-served order.

If a Resource unit is free when the request is made, the requesting process takes it and moves on to the next statement in its PEM. If no Resource unit is available when the request is made, then the requesting process is appended to the Resource's `waitQ` and suspended. The next time a unit becomes available the first process in the `r.waitQ` takes it and continues its execution. All priority assignments are ignored. Moreover, in the FIFO case no preemption is possible, for preemption requires that priority assignments be recognized. (However, see the *Note on preemptive requests with waitQ in FIFO order* for one way of simulating such situations.)

Example In this complete script, the `server` Resource object is given two resource units (`capacity=2`). By not specifying its `Qtype` it takes the default value, `FIFO`. Here six clients arrive in the order specified by the program. They all request a resource unit from the `server` Resource object at the same time. Even though they all specify a priority value in their requests, it is ignored and they get their Resource units in the same order as their requests:

```
from SimPy.Simulation import (Process, Resource, activate, initialize, hold,
                              now, release, request, simulate)


class Client(Process):
```

```
    inClients = []      # list the clients in order by their requests
    outClients = []     # list the clients in order by completion of service

    def __init__(self, name):
        Process.__init__(self, name)

    def getserved(self, servtime, priority, myServer):
        Client.inClients.append(self.name)
        print('%s requests 1 unit at t = %s' % (self.name, now()))
        # request use of a resource unit
        yield request, self, myServer, priority
        yield hold, self, servtime
        # release the resource
        yield release, self, myServer
        print('%s done at t = %s' % (self.name, now()))
        Client.outClients.append(self.name)


initialize()

# the next line creates the ``server`` Resource object
server = Resource(capacity=2)  # server defaults to qType==FIFO

# the next lines create some Client process objects
c1, c2 = Client(name='c1'), Client(name='c2')
c3, c4 = Client(name='c3'), Client(name='c4')
c5, c6 = Client(name='c5'), Client(name='c6')

# in the next lines each client requests
# one of the ``server``'s Resource units
activate(c1, c1.getserved(servtime=100, priority=1, myServer=server))
activate(c2, c2.getserved(servtime=100, priority=2, myServer=server))
activate(c3, c3.getserved(servtime=100, priority=3, myServer=server))
activate(c4, c4.getserved(servtime=100, priority=4, myServer=server))
activate(c5, c5.getserved(servtime=100, priority=5, myServer=server))
activate(c6, c6.getserved(servtime=100, priority=6, myServer=server))

simulate(until=500)

print('Request order: %s' % Client.inClients)
print('Service order: %s' % Client.outClients)
```

This program results in the following output:

```
c1 requests 1 unit at t = 0
c2 requests 1 unit at t = 0
c3 requests 1 unit at t = 0
c4 requests 1 unit at t = 0
c5 requests 1 unit at t = 0
c6 requests 1 unit at t = 0
c1 done at time = 100
c2 done at time = 100
c3 done at time = 200
c4 done at time = 200
c5 done at time = 300
c6 done at time = 300

Request order:  ['c1', 'c2', 'c3', 'c4', 'c5', 'c6']
```

```
Service order:  ['c1', 'c2', 'c3', 'c4', 'c5', 'c6']
```

As illustrated, the clients are served in FIFO order. Clients `c1` and `c2` each take one Resource unit right away, but the others must wait. When `c1` and `c2` finish with their resources, clients `c3` and `c4` can each take a unit, and so forth.

Priority requests for a Resource unit

If the Resource `r` is defined with `qType==PriorityQ`, priority values in requests are recognized. If a Resource unit is available when the request is made, the requesting process takes it. If no Resource unit is available when the request is made, the requesting process is inserted into the Resource's `waitQ` in order of priority (from high to low) and suspended. For an example where priorities are used, we simply change the preceding example's specification of the `server` Resource object to:

```
server = Resource(capacity=2, qType=PriorityQ)
```

where, by not specifying it, we allow `preemptable` to take its default value, `False`.

Example After this change the program's output becomes:

```
c1 requests 1 unit at t = 0
c2 requests 1 unit at t = 0
c3 requests 1 unit at t = 0
c4 requests 1 unit at t = 0
c5 requests 1 unit at t = 0
c6 requests 1 unit at t = 0
c1 done at time = 100
c2 done at time = 100
c6 done at time = 200
c5 done at time = 200
c4 done at time = 300
c3 done at time = 300

Request order:  ['c1', 'c2', 'c3', 'c4', 'c5', 'c6']
Service order:  ['c1', 'c2', 'c6', 'c5', 'c4', 'c3']
```

Although `c1` and `c2` have the lowest priority values, each requested and got a `server` unit immediately. That was because at the time they made those requests a `server` unit was available and the `server.waitQ` was empty – it did not start to fill until `c3` made its request and found all of the `server` units busy. When `c1` and `c2` completed service, `c6` and `c5` (with the highest priority values of all processes in the `waitQ`) each got a Resource unit, etc.

When some processes in the `waitQ` have the same priority level as a process making a priority request, SimPy inserts the requesting process immediately *behind* them. Thus for a given priority value, processes are placed in FIFO order. For example, suppose that when a "priority 3" process makes its priority request the current `waitQ` consists of processes with priorities `[5,4,3a,3b,3c,2a,2b,1]`, where the letters indicate the order in which the equal-priority processes were placed in the queue. Then SimPy inserts this requesting process into the current `waitQ` immediately behind its last "priority 3" process. Thus, the new `waitQ` will be `[5,4,3a,3b,3c,3d,2a,2b,1]`, where the inserted process is `3d`.

One consequence of this is that, if all priority requests are assigned the same priority value, then the `waitQ` will in fact be maintained in `FIFO` order. In that case, using a `FIFO` instead of a `PriorityQ` discipline provides some saving in execution time which may be important in simulations where the `waitQ` may be long.

Preemptive requests for a Resource unit

In some models, higher priority processes can actually *preempt* lower priority processes, i.e., they can take over and use a Resource unit currently being used by a lower priority process whenever no free Resource units are available. A Resource object that allows its units to be preempted is created by setting its properties to `qType == PriorityQ` and `preemptable == True`.

Whenever a `preemptable` Resource unit is free when a request is made, then the requesting process takes it and continues its execution. On the other hand, when a higher priority request finds all the units in a `preemptable` Resource in use, then SimPy adopts the following procedure regarding the Resource's `activeQ` and `waitQ`:

- The process with the lowest priority is removed from the `activeQ`, suspended, and put at the front of the `waitQ` – so (barring additional preemptions) it will be the next one to get a resource unit.
- The preempting process gets the vacated resource unit and is inserted into the `activeQ` in order of its priority value.
- The time for which the preempted process had the resource unit is taken into account when the process gets into the `activeQ` again. Thus, its *total hold time* is always the same, regardless of how many times it has been preempted.

Warning: SimPy only supports preemption of processes which are implemented in the following pattern:

```
yield request (one or more request statements)
<some code>
yield hold (one or more hold statements)
<some code>
yield release (one or more release statements)
```

Modelling the preemption of a process in any other pattern may lead to errors or exceptions.

We emphasize that a process making a `preemptive` request to a fully-occupied Resource gets a resource unit if – but only if – some process in the current `activeQ` has a lower priority. Otherwise, it will be inserted into the `waitQ` at a location determined by its priority value and the current contents of the `waitQ`, using a procedure analogous to that described for priority requests near the end of the preceding section on *Priority requests for a Resource unit*. This may have the effect of advancing the preempting process ahead of any lower-priority processes that had earlier been preempted and put at the head of the `waitQ`. In fact, if several preemptions occur before a unit of resource is freed up, then the head of the `waitQ` will consist of the processes that have been preempted – in order from the last process preempted to the first of them.

Example In this example two clients of different priority compete for the same resource unit:

```
from SimPy.Simulation import (PriorityQ, Process, Resource, activate,
                              initialize, hold, now, release, request,
                              simulate)


class Client(Process):
    def __init__(self, name):
        Process.__init__(self, name)

    def getserved(self, servtime, priority, myServer):
        print('%s requests 1 unit at t=%s' % (self.name, now()))
        yield request, self, myServer, priority
        yield hold, self, servtime
        yield release, self, myServer
        print('%s done at t=%s' % (self.name, now()))
```

```
initialize()
# create the *server* Resource object
server = Resource(capacity=1, qType=PriorityQ, preemptable=1)
# create some Client process objects
c1 = Client(name='c1')
c2 = Client(name='c2')
activate(c1, c1.getserved(servtime=100, priority=1, myServer=server), at=0)
activate(c2, c2.getserved(servtime=100, priority=9, myServer=server), at=50)
simulate(until=500)
```

The output from this program is:

```
c1 requests 1 unit at t=0
c2 requests 1 unit at t=50
c2 done at t=150
c1 done at t=200
```

Here, `c1` is preempted by `c2` at `t=50`. At that time, `c1` had held the resource for 50 of its total of 100 time units. When `c2` finished and released the resource unit at 150, `c1` got the resource back and finished the last 50 time units of its service at `t=200`.

If preemption occurs when the last few processes in the current `activeQ` have the same priority value, then the last process in the current `activeQ` is the one that will be preempted and inserted into the `waitQ` ahead of all others. To describe this, it will be convenient to indicate by an added letter the order in which equal-priority processes have been inserted into a queue. Now, suppose that a "priority 4" process makes a preemptive request when the current `activeQ` priorities are `[5,3a,3b]` and the current `waitQ` priorities are `[2,1,0a,0b]`. Then process `3b` will be preempted. After the preemption the `activeQ` will be `[5,4,3a]` and the `waitQ` will be `[3b,2,1,0a,0b]`.

Note on preemptive requests with waitQ in FIFO order

You may consider doing the following to model a system whose queue of items waiting for a resource is to be maintained in FIFO order, but in which preemption is to be possible. It uses SimPy's `preemptable` Resource objects, and uses priorities in a way that allows for preempts while maintaining a FIFO `waitQ` order.

- Set `qType=PriorityQ` and `preemptable=True` (so that SimPy will process preemptive requests correctly).
- Model "system requests that are to be considered as non-preemptive" in SimPy as process objects each of which has exactly the same (low) priority value – for example, either assign all of them a priority value of 0 (zero) or let it default to that value. (This has the effect of maintaining all of these process objects in the `waitQ` in FIFO order, as explained at the end of the section on *Priority requests for a Resource unit*, above.)
- Model "system requests that are to be considered as preemptive" in SimPy as process objects each of which is assigned a uniform priority value, but give them a higher value than the one used to model the "non-preemptive system requests" – for example, assign all of them a priority value of 1 (one). Then they will have a higher priority value than any of the non-preemptive requests.

Example Here is an example of how this works for a Resource with two Resource units – we give the `activeQ` before the `waitQ` throughout this example:

1. Suppose that the current `activeQ` and `waitQ` are `[0a,0b]` and `[0c]`, respectively.

2. A "priority 1" process makes a preemptive request. Then the queues become: `[1a,0a]` and‘‘ [0b,0c]‘‘.
3. Another "priority 1" process makes a preemptive request. Then the queues become: `[1a,1b]` and `[0a,0b,0c]`.
4. A third "priority 1" process makes a preemptive request. Then the queues become: `[1a,1b]` and `[1c,0a,0b,0c]`.
5. Process `1a` finishes using its resource unit. Then the queues become: `[1b,1c]` and `[0a,0b,0c]`.

Reneging – leaving a queue before acquiring a resource

In most real world situations, people and other items do not wait forever for a requested resource facility to become available. Instead, they leave its queue when their patience is exhausted or when some other condition occurs. This behaviour is called *reneging*, and the reneging person or thing is said to *renege*.

SimPy provides an extended (i.e., compound) `yield request` statement to handle reneging.

Reneging yield request

There are two types of reneging clause, one for reneging after a certain time and one for reneging when an event has happened. Their general form is

`yield (request,self,`*r [,P]*`),(`*<reneging clause>*`)`

to request a unit of Resource *r* (with optional priority *P*, assuming the Resource has been defined as a `priorityQ`) but with reneging.

A SimPy program that models Resource requests with reneging must use the following pattern of statements:

```
yield (request, self, r), (<reneging clause>)
if self.acquired(resource):
   ## process got resource and so did NOT renege
   . . . .
   yield release, self, resource
else:
   ## process reneged before acquiring resource
   . . . . .
```

A call to the `self.acquired(resource)` method is mandatory after a compound `yield request` statement. It not only indicates whether or not the process has acquired the resource, it also removes the reneging process from the resource's `waitQ`.

Reneging after a time limit

To make a process give up (renege) after a certain time, use a reneging clause of the following form:

`yield (request,self,`*r [,P]*`),(hold, self, ``\ *waittime*\ ``)`

Here the process requests one unit of the resource *r* with optional priority *P*. If a resource unit is available it takes it and continues its PEM. Otherwise, as usual, it is passivated and inserted into *r*'s `waitQ`.

The process takes a unit if it becomes available before *waittime* expires and continues executing its PEM. If, however, the process has not acquired a unit before the *waittime* has expired it abandons the request (reneges) and leaves the `waitQ`.

Example: part of a parking lot simulation:

```
. . . .
parking_lot = Resource(capacity=10)
patience = 5     # wait no longer than "patience" time units
                 # for a parking space
park_time = 60   # park for "park_time" time units if get a parking space
. . . .
yield (request, self, parking_lot), (hold, self, patience)
if self.acquired(parking_lot):
   # park the car
   yield hold, self, park_time
   yield release, self, parking_lot
else:
   # patience exhausted, so give up
   print("I'm not waiting any longer. I am going home now.")
```

Reneging when an event has happened

To make a process renege at the occurrence of an event, use a reneging clause having a pattern like the one used for a `yield waitevent` statement, namely `waitevent,self,`*events* (see *yield waitevent*). For example:

`yield (request,self,`*r [,P]*`),(waitevent,self,`*events*`)`

Here the process requests one unit of the resource *r* with optional priority *P*. If a resource unit is available it takes it and continues its PEM. Otherwise, as usual, it is passivated and inserted into *r*'s `waitQ`.

The process takes a unit if it becomes available before any of the *events* occur, and continues executing its PEM. If, however, any of the SimEvents in *events* occur first, it abandons the request (reneges) and leaves the `waitQ`. (Recall that *events* can be either one event, a list, or a tuple of several SimEvents.)

Example Queuing for movie tickets (part):

```
. . . .
seats = Resource(capacity=100)
sold_out = SimEvent() # signals "out of seats"
too_late = SimEvent() # signals "too late for this show"
. . . .
# Leave the ticket counter queue when movie sold out
# or it is too late for the show
yield (request, self, seats), (waitevent, self, [sold_out, too_late])
if self.acquired(seats):
   # watch the movie
   yield hold, self, 120
   yield release, self, seats
else:
   # did not get a seat
   print('Who needs to see this silly movie anyhow?')
```

Exiting conventions and preemptive queues

Many discrete event simulations (including SimPy) adopt the normal "exiting convention", according to which processes that have once started using a Resource unit stay in some Resource queue until their `hold` time has completed. This is of course automatically the case for FIFO and non-preemptable `PriorityQ` disciplines. The point is that the exiting convention is also applied in the `preemptable` queue discipline case. Thus, processes remain in some Resource queue until their `hold` time has completed, even if they are preempted by higher priority processes.

Some real-world situations conform to this convention and some do not. An example of one that does conform can be described as follows. Suppose that at work you are assigned tasks of varying levels of priority. You are to set aside lower priority tasks in order to work on higher priority ones. But you are eventually to complete all of your assigned tasks. So you are operating like a SimPy resource that obeys a `preemptable` queue discipline and has one resource unit. With this convention, half-finished low-priority tasks may be postponed indefinitely if they are continually preempted by higher-priority tasks.

An example that does not conform to the exiting convention can be described as follows. Suppose again that you are assigned tasks of varying levels of priority and are to set aside lower priority tasks to work on higher priority ones. But you are instructed that any tasks not completed within 24 hours after being assigned are to be sent to another department for completion. Now, suppose that you are assigned Task-A that has a priority level of 3 and will take 10 hours to complete. After working on Task-A for an hour, you are assigned Task-B, which has a priority level of 5 and will take 20 hours to complete. Then, at 11 hours, after working on Task-B for 10 hours, you are assigned Task-C, which has a priority level of 1 and will take 4 hours to complete. (At this point Task-B needs 10 hours to complete, Task-A needs 9 hours to complete, and Task-C needs 4 hours to complete.) At 21 hours you complete Task-B and resume working on Task-A, which at that point needs 9 hours to complete. At 24 hours Task-A still needs another 6 hours to complete, but it has reached the 24-hour deadline and so is sent to another department for completion. At the same time, Task-C has been in the waitQ for 13 hours, so you take it up and complete it at hour 28. This queue discipline does not conform to the exiting convention, for under that convention at 24 hours you would continue work on Task-A, complete it at hour 30, and then start on Task-C.

Recording Resource queue lengths

Many discrete event models are used mainly to explore the statistical properties of the `waitQ` and `activeQ` associated with some or all of their simulated resources. SimPy's support for this includes the *Monitor* and the *Tally*. For more information on these and other recording methods, see the section on *Recording Simulation Results*.

If a Resource, `r`, is defined with `monitored=True` SimPy automatically records the length of its associated `waitQ` and `activeQ`. These records are kept in the recorder objects called `r.waitMon` and `r.actMon`, respectively. This solves a problem, particularly for the `waitQ` which cannot easily be recorded externally to the resource.

The property `monitorType` indicates which variety of recorder is to be used, either *Monitor* or *Tally*. The default is `Monitor`. If this is chosen, complete time series for both queue lengths are maintained and can be used for advanced post-simulation statistical analyses as well as for displaying summary statistics (such as averages, standard deviations, and histograms). If `Tally` is chosen summary statistics can be displayed, but complete time series cannot. For more information on these and SimPy's other recording methods, see the section on *Recording Simulation Results*.

Example The following program uses a `Monitor` to record the `server` resource's queues. After the simulation ends, it displays some summary statistics for each queue, and then their complete time series:

```
from math import sqrt
from SimPy.Simulation import (Monitor, Process, Resource, activate, initialize,
                              hold, now, release, request, simulate)


class Client(Process):
```

```
    inClients = []
    outClients = []

    def __init__(self, name):
        Process.__init__(self, name)

    def getserved(self, servtime, myServer):
        print('%s requests 1 unit at t = %s' % (self.name, now()))
        yield request, self, myServer
        yield hold, self, servtime
        yield release, self, myServer
        print('%s done at t = %s' % (self.name, now()))


initialize()

server = Resource(capacity=1, monitored=True, monitorType=Monitor)

c1, c2 = Client(name='c1'), Client(name='c2')
c3, c4 = Client(name='c3'), Client(name='c4')

activate(c1, c1.getserved(servtime=100, myServer=server))
activate(c2, c2.getserved(servtime=100, myServer=server))
activate(c3, c3.getserved(servtime=100, myServer=server))
activate(c4, c4.getserved(servtime=100, myServer=server))

simulate(until=500)

print('')
print('(TimeAverage no. waiting: %s' % server.waitMon.timeAverage())
print('(Number) Average no. waiting: %.4f' % server.waitMon.mean())
print('(Number) Var of no. waiting: %.4f' % server.waitMon.var())
print('(Number) SD of no. waiting: %.4f' % sqrt(server.waitMon.var()))
print('(TimeAverage no. in service: %s' % server.actMon.timeAverage())
print('(Number) Average no. in service: %.4f' % server.actMon.mean())
print('(Number) Var of no. in service: %.4f' % server.actMon.var())
print('(Number) SD of no. in service: %.4f' % sqrt(server.actMon.var()))
print('=' * 40)
print('Time history for the "server" waitQ:')
print('[time, waitQ]')
for item in server.waitMon:
    print(item)
print('=' * 40)
print('Time history for the "server" activeQ:')
print('[time, activeQ]')
for item in server.actMon:
    print(item)
```

The output from this program is:

```
c1 requests 1 unit at t = 0
c2 requests 1 unit at t = 0
c3 requests 1 unit at t = 0
c4 requests 1 unit at t = 0
c1 done at t = 100
c2 done at t = 200
c3 done at t = 300
c4 done at t = 400
```

```
(TimeAverage no. waiting: 1.5
(Number) Average no. waiting: 1.2857
(Number) Var of no. waiting: 1.0612
(Number) SD of no. waiting: 1.0302
(TimeAverage no. in service: 1.0
(Number) Average no. in service: 0.4444
(Number) Var of no. in service: 0.2469
(Number) SD of no. in service: 0.4969
========================================
Time history for the "server" waitQ:
[time, waitQ]
[0, 0]
[0, 1]
[0, 2]
[0, 3]
[100, 2]
[200, 1]
[300, 0]
========================================
Time history for the "server" activeQ:
[time, activeQ]
[0, 0]
[0, 1]
[100, 0]
[100, 1]
[200, 0]
[200, 1]
[300, 0]
[300, 1]
[400, 0]
```

This output illustrates the difference between the *(Time) Average* and the *number statistics*. Here process `c1` was in the `waitQ` for zero time units, process `c2` for 100 time units, and so forth. The total wait time accumulated by all four processes during the entire simulation run, which ended at time 400, amounts to 0 + 100 + 200 + 300 = 600 time units. Dividing the 600 accumulated time units by the simulation run time of 400 gives 1.5 for the *(Time) Average* number of processes in the `waitQ`. It is the time-weighted average length of the `waitQ`, but is almost always called simply the average length of the `waitQ` or the average number of items waiting for a resource.

It is also the expected number of processes you would find in the `waitQ` if you took a snapshot of it at a random time during the simulation. The `activeQ`'s time average computation is similar, although in this example the resource is held by some process throughout the simulation. Even though the number in the `activeQ` momentarily drops to zero as one process releases the resource and immediately rises to one as the next process acquires it, that occurs instantaneously and so contributes nothing to the *(Time) Average* computation.

Number statistics such as the Average, Variance, and Standard Deviation are computed differently. At time zero the number of processes in the `waitQ` starts at 1, then rises to 2, and then to 3. At time 100 it drops back to two processes, and so forth. The average and standard deviation of the six values [1, 2, 3, 2, 1, 0] is 1.5 and 0.9574..., respectively. Number statistics for the `activeQ` are computed using the eight values [1, 0, 1, 0, 1, 0, 1, 0] and are as shown in the output.

When the `monitorType` is changed to `Tally`, all the output up to and including the lines:

```
Time history for the 'server' waitQ:
[time, waitQ]
```

is displayed. Then the output concludes with an error message indicating a problem with the reference to `server.waitMon`. Of course, this is because `Tally` does not generate complete time series.

[Return to *Top*]

2.1.5 Levels

The three resource facilities provided by the SimPy system are *Resources*, *Levels* and *Stores*. Each models a congestion point where process objects may have to queue up to obtain resources. This section describes the Level type of resource facility.

Levels model the production and consumption of a homogeneous undifferentiated "material." Thus, the currently-available amount of material in a Level resource facility can be fully described by a scalar (real or integer). Process objects may increase or decrease the currently-available amount of material in a Level facility.

For example, a gasoline station stores gas (petrol) in large tanks. Tankers increase, and refuelled cars decrease, the amount of gas in the station's storage tanks. Both getting amounts and putting amounts may be subjected to *reneging* like requesting amounts from a Resource.

Defining a Level

You define the Level resource facility *lev* by a statement like this:

```
lev = Level(name='a_level', unitName='units',
            capacity='unbounded', initialBuffered=0,
            putQType=FIFO, getQType=FIFO,
            monitored=False, monitorType=Monitor)
```

where

- `name` (string type) is a descriptive name for the Level object *lev* is known (e.g., `'inventory'`).
- `unitName` (string type) is a descriptive name for the units in which the amount of material in *lev* is measured (e.g., `'kilograms'`).
- `capacity` (positive real or integer) is the capacity of the Level object *lev*. The default value is set to `'unbounded'` which is interpreted as `sys.maxint`.
- `initialBuffered` (positive real or integer) is the initial amount of material in the Level object *lev*.
- `putQType` (`FIFO` or `PriorityQ`) is the (producer) queue discipline.
- `getQType` (`FIFO` or `PriorityQ`) is the (consumer) queue discipline.
- `monitored` (boolean) specifies whether the queues and the amount of material in *lev* will be recorded.
- `monitorType` (`Monitor` or `Tally`) specifies which type of *Recorder* to use. Defaults to `Monitor`.

Every Level resource object, such as `lev`, also has the following additional attributes:

- `lev.amount` is the amount currently held in *lev*.
- `lev.putQ` is the queue of processes waiting to add amounts to *lev*, so `len(lev.putQ)` is the number of processes waiting to add amounts.
- `lev.getQ` is the queue of processes waiting to get amounts from *lev*, so `len(lev.getQ)` is the number of processes waiting to get amounts.
- `lev.monitored` is `True` if the queues are to be recorded. In this case `lev.putQMon`, `lev.getQMon`, and `lev.bufferMon` exist.

- `lev.putQMon` is a *Recorder* observing `lev.putQ`.
- `lev.getQMon` is a *Recorder* observing `lev.getQ`.
- `lev.bufferMon` is a *Recorder* observing `lev.amount`.

Getting amounts from a Level

Processes can request amounts from a Level and the same or other processes can offer amounts to it.

A process, the *requester*, can request an amount *ask* from the Level resource object *lev* by a `yield get` statement.:

- `yield get,self,` *lev,ask[,P]*

Here *ask* must be a positive real or integer (the amount) and *P* is an optional priority value (real or integer). If *lev* does not hold enough to satisfy the request (that is, *ask* > *lev*`.amount`) the requesting process is passivated and queued (in *lev*`.getQ`) in order of its priority. Subject to the priority order, it will be reactivated when there is enough to satisfy the request.

`self.got` holds the amount actually received by the requester.

..index:: Level; put

Putting amounts into a Level

A process, the *offerer*, which is usually but not necessarily different from the *requester*, can offer an amount *give* to a Level, *lev*, by a `yield put` statement:

- `yield put, self,` *lev,give[,P]*

Here *give* must be a positive real or integer, and *P* is an optional priority value (real or integer). If the amount offered would lead to an overflow (that is, *lev*.`amount` + *give* > *lev*.`capacity`) the offering process is passivated and queued (in `lev.putQ`). Subject to the priority order, it will be reactivated when there is enough space to hold the amount offered.

The orderings of processes in a Level's `getQ` and `putQ` behave like those described for the `waitQ` under *Resources*, except that they are not preemptable. Thus, priority values are ignored when the queue type is FIFO. Otherwise higher priority values have higher priority, etc.

Example. Suppose that a random demand on an inventory is made each day. Each requested amount is distributed normally with a mean of 1.2 units and a standard deviation of 0.2 units. The inventory (modelled as an object of the Level class) is refilled by 10 units at fixed intervals of 10 days. There are no back-orders, but a accumulated sum of the total stock-out quantities is to be maintained. A trace is to be printed out each day and whenever there is a stock-out:

```
from random import normalvariate, seed
from SimPy.Simulation import (Level, Process, activate, get, initialize, hold,
                              now, put, simulate)


class Deliver(Process):
    def deliver(self):  # an "offeror" PEM
        while True:
            lead = 10.0      # time between refills
            delivery = 10.0  # amount in each refill
            yield put, self, stock, delivery
            print('at %6.4f, add %6.4f units, now amount = %6.4f' %
                  (now(), delivery, stock.amount))
```

```
            yield hold, self, lead


class Demand(Process):
    stockout = 0.0      # initialize initial stockout amount

    def demand(self):  # a "requester" PEM
        day = 1.0       # set time-step to one day
        while True:
            yield hold, self, day
            dd = normalvariate(1.20, 0.20)  # today's random demand
            ds = dd - stock.amount
            # excess of demand over current stock amount
            if dd > stock.amount:     # can't supply requested amount
                yield get, self, stock, stock.amount
                # supply all available amount
                self.stockout += ds
                # add unsupplied demand to self.stockout
                print('day %6.4f, demand = %6.4f, shortfall = %6.4f' %
                      (now(), dd, -ds))
            else:  # can supply requested amount
                yield get, self, stock, dd
                print('day %6.4f, supplied %6.4f, now amount = %6.4f' %
                      (now(), dd, stock.amount))


stock = Level(monitored=True)     # 'unbounded' capacity and other defaults

seed(99999)
initialize()

offeror = Deliver()
activate(offeror, offeror.deliver())
requester = Demand()
activate(requester, requester.demand())

simulate(until=49.9)

result = (stock.bufferMon.mean(), requester.stockout)
print('')
print('Summary of results through end of day %6.4f:' % int(now()))
print('average stock = %6.4f, cumulative stockout = %6.4f' % result)
```

Here is the last ten day's output from one run of this program:

```
at 40.0000, add 10.0000 units, now amount = 10.0000
day 40.0000, supplied  0.7490, now amount = 9.2510
day 41.0000, supplied  1.1651, now amount = 8.0858
day 42.0000, supplied  1.1117, now amount = 6.9741
day 43.0000, supplied  1.1535, now amount = 5.8206
day 44.0000, supplied  0.9202, now amount = 4.9004
day 45.0000, supplied  0.8990, now amount = 4.0014
day 46.0000, supplied  1.1448, now amount = 2.8566
day 47.0000, supplied  1.7287, now amount = 1.1279
day 48.0000, supplied  0.9608, now amount = 0.1670
day 49.0000, demand =  0.9837, shortfall = -0.8167

Summary of results through end of day 49.0000:
```

```
14 average stock =  4.2720, cumulative stockout =  9.7484
```

[Return to *Top*]

Reneging

The `yield put` can be subject to *reneging* using one of the compound statements:

- `yield (put,self,`*lev,ask[,P]*`),(hold,self,`*waittime*`)`

where if the process does not acquire the amount before *waittime* is elapsed, the offerer leaves the `waitQ` and its execution continues or

- `yield (put,self,`*lev,ask[,P]*`),(waitevent,self,`*events*`)`

where if one of the SimEvents in *events* occurs before enough becomes available, the offerer leaves the `waitQ` and its execution continues.

In either case if reneging has *not* occurred the quantity will have been put into the Level and `self.stored(`*lev*`)` will be `True`. This must be tested immediately after the `yield`:

```
1 yield (put, self, lev, ask[,P]), (<reneging clause>)
2 if self.stored(lev):
3    ## process  did not renege
4    . . . .
5 else:
6    ## process reneged before being able to put into the resource
```

The `yield get` can also be subject to *reneging* using one of the compound statements:

- `yield (get, self, lev, ask[,P]), (hold, self, waittime)`

where if the process does not acquire the amount before *waittime* is elapsed, the offerer leaves the `waitQ` and its execution continues.

- `yield (get,self,lev,ask[,P]),(waitevent,self,events)`

where if one of the SimEvents in *events* occurs before enough becomes available, reneging occurs, the offerer leaves the `waitQ` and its execution continues.

In either case if reneging has *not* occurred `self.got == ask` and `self.acquired(lev)` will be `True`. `self.acquired(lev)` must be called immediately after the `yield`:

```
1 yield (get, self, lev, ask[,P]), (<reneging clause>)
2 if self.acquired(lev):
3    ## process  did not renege, self.got == ask
4    . . . .
5 else:
6    ## process reneged before being able to put into the resource
```

This test removes the reneging process from the `getQ`.

[Return to *Top*]

2.1.6 Stores

The three resource facilities provided by the SimPy system are *Resources*, *Levels* and *Stores*. Each models a congestion point where process objects may have to queue up to obtain resources. This section describes the Store type of resource facility.

Stores model the production and consumption of individual items of any Python type. Process objects can insert or remove specific items from the list of items available in a Store. For example, surgical procedures (treated as process objects) require specific lists of personnel and equipment that may be treated as the items available in a Store type of resource facility such as a clinic or hospital. As the items held in a Store may be of any Python type, they may in particular be process objects, and this can be exploited to facilitate modelling Master/Slave relationships. *putting* and *getting* may also be subjected to reneging.

Defining a Store

The Store object `sObj` is established by a statement like the following:

```
sObj = Store(name='a_store',
             unitName='units',
             capacity='unbounded',
             initialBuffered=None,
             putQType=FIFO,
             getQType=FIFO,
             monitored=False,
             monitorType=Monitor)
```

where

- `name` (string type) is a descriptive name for *sObj* (e.g., `'Inventory'`).
- `unitName` (string type) is a descriptive name for the items in *sObj* (e.g., `'widgets'`).
- `capacity` (positive integer) is the maximum number of individual items that can be held in *sObj*. The default value is set to `'unbounded'` which is interpreted as `sys.maxint`.
- `initialBuffered` (a list of individual items) is *sObj*'s initial content.
- `putQType` (`FIFO` or `PriorityQ`) is the (producer) queue discipline.
- `getQType` (`FIFO` or `PriorityQ`) is the (consumer) queue discipline.
- `monitored` (boolean) specifies whether *sObj*'s queues and contents are to be recorded.
- `monitorType` (`Monitor` or `Tally`) specifies the type of *Recorder* to be used. Defaults to `Monitor`.

The Store object *sObj* also has the following additional attributes:

- `sObj.theBuffer` is a queue (list) of the individual items in *sObj*. This list is in FIFO order unless the user stores them in a particular order (see *Storing objects in an order* , below). It is read-only and not directly changeable by the user.
- `sObj.nrBuffered` is the current number of objects in *sObj*. This is read-only and not directly changeable by the user.
- `sObj.putQ` is the queue of processes waiting to add items to *sObj*, so that `len(sObj.putQ)` is the number of processes waiting to add items.
- `sObj.getQ` is the queue of processes waiting to get items from *sObj*, so that `len(sObj.getQ)` is the number of processes waiting to get items.
- If `sObj.monitored` is `True` then the queues are to be recorded. In this case `sObj.putQMon`, `sObj.getQMon`, and `sObj.bufferMon` exist.

- `sObj.putQMon` is a *Recorder* observing `sObj.putQ`.
- `sObj.getQMon` is a *Recorder* observing `sObj.getQ`.
- `sObj.bufferMon` is a *Recorder* observing `sObj.nrBuffered`.

Putting objects into a Store

Processes can request items from a Store and the same or other processes can offer items to it. First look at the simpler of these operations, the `yield put`.

A process, the *offerer*, which is usually but not necessarily different from the *requester*, can offer a list of items to *sObj* by a `yield put` statement:

- `yield put, self, sObj, give[,P]`

Here `give` is a list of any Python objects. If this statement would lead to an overflow (that is, `sObj.nrBuffered + len(give) > sObj.capacity`) the putting process is passivated and queued (in `sObj.putQ`) until there is sufficient room. *P* is an optional priority value (real or integer).

The ordering of processes in a Store's putQ and getQ behave like those described for the waitQ under *Resources* , except that they are not preemptable. Thus, priority values are ignored when the queue type is FIFO. Otherwise higher priority values indicate higher priority, etc.

The items in *sObj* are stored in the form of a queue called `sObj.theBuffer`, which is in FIFO order unless the user has arranged to sort them into a particular order (see *Storing objects in an order* below).

Getting objects from a Store

There are two ways of getting objects from a Store. A process, the *requester*, can either extract the first `n` objects from *sObj* or a list of items chosen by a *filter function*.

Getting *n* items is achieved by the following statement:

- `yield get, self, sObj, n [,P]`

Here *n* must be a positive integer and *P* is an optional priority value (real or integer). If *sObj* does not currently hold enough objects to satisfy this request (that is, `n > sObj.nrBuffered`) then the requesting process is passivated and queued (in `sObj.getQ`). Subject to the priority ordering, it will be reactivated when the request can be satisfied.

The retrieved objects are returned in the list attribute `got` of the requesting process.

`yield get` requests with a numerical parameter are honored in priority/FIFO order. Thus, if there are two processes in the Store's `getQ`, with the first requesting two items and the second one, the second process gets the requested item only after the first process has been given its two items.

Using the get filter function

The second method is to get a list of items chosen by a *filter function*, written by the user.

The command, using filter function *ffn* is as follows:

- `yield get, self, sObj, ffn [,P]`

The user provides a filter function that has a single list argument and returns a list. The argument represents the buffer of the Store. The function must search through the objects in the buffer and return a sub-list of those that satisfy the requirement.

Example The filter function `allweight`, shown below, is an example of such a filter. The argument, `buff`, will be automatically replaced in the execution of `yield get,self,store,allweight` by the buffer of the Store. In this example the objects in the Store are assumed to have `weight` attributes. The function `allweight` selects all those that have a weight attribute over a value `W` and returns these as a list. The list appears to the calling process as `self.got`:

```
def allweight(buff):
    """filter: get all items with .weight >=W from store"""
    result = []
    for i in buff:
        if i.weight >= W:
            result.append(i)
    return result
```

This might be used as follows:

```
yield get, self, sObj, allweight [,P]
```

The retrieved objects are returned in the list attribute `got` of the requesting process.

Note: *''yield get''* requests with a filter function parameter are not necessarily honored in priority/FIFO order, but rather according to the filter function. An example: There are two processes in the Store's `getQ`, with the first requesting an item with a *weight* attribute less than 2 kilograms and the second one requesting one with a `weight` attribute less than 3 kilograms. If there is an item in the Store's buffer with a `weight` attribute between 2 and 3 and none with an attribute of less than 2, the second `get` requester gets unblocked before the first one. Effectively, the SimPy run time system runs through all processes in the `getQ` in sequence and tests their filter functions as long as there are still items in the Store's buffer.

Example The following program illustrates the use of a Store to model the production and consumption of "widgets". The widgets are distinguished by their weight:

```
from SimPy.Simulation import (Lister, Process, Store, activate, get, hold,
                              initialize, now, put, simulate)


class ProducerD(Process):
    def __init__(self):
        Process.__init__(self)

    def produce(self):  # the ProducerD PEM
        while True:
            yield put, self, buf, [Widget(9), Widget(7)]
            yield hold, self, 10


class ConsumerD(Process):
    def __init__(self):
        Process.__init__(self)

    def consume(self):  # the ConsumerD PEM
        while True:
            toGet = 3
            yield get, self, buf, toGet
            assert len(self.got) == toGet
            print('%s Get widget weights %s' % (now(),
```

```
                                                [x.weight for x in self.got]))
            yield hold, self, 11


class Widget(Lister):
    def __init__(self, weight=0):
        self.weight = weight


widgbuf = []
for i in range(10):
    widgbuf.append(Widget(5))

initialize()

buf = Store(capacity=11, initialBuffered=widgbuf, monitored=True)

for i in range(3):  # define and activate 3 producer objects
    p = ProducerD()
    activate(p, p.produce())

for i in range(3):  # define and activate 3 consumer objects
    c = ConsumerD()
    activate(c, c.consume())

simulate(until=50)

print('LenBuffer: %s' % buf.bufferMon)  # length of buffer
print('getQ: %s' % buf.getQMon)         # length of getQ
print('putQ %s' % buf.putQMon)          # length of putQ
```

This program produces the following outputs (some lines may be formatted differently):

```
0 Get widget weights [5, 5, 5]
0 Get widget weights [5, 5, 5]
0 Get widget weights [5, 5, 5]
11 Get widget weights [5, 9, 7]
11 Get widget weights [9, 7, 9]
11 Get widget weights [7, 9, 7]
22 Get widget weights [9, 7, 9]
22 Get widget weights [7, 9, 7]
22 Get widget weights [9, 7, 9]
33 Get widget weights [7, 9, 7]
33 Get widget weights [9, 7, 9]
40 Get widget weights [7, 9, 7]
44 Get widget weights [9, 7, 9]
50 Get widget weights [7, 9, 7]
LenBuffer: [[0, 10], [0, 7], [0, 9], [0, 11], [0, 8], [0, 10], [0, 7], [10, 9], [10,␣
↪11], [11, 8], [11, 10], [11, 7], [11, 4], [20, 6], [20, 8], [21, 10], [22, 7], [22,␣
↪4], [22, 1], [30, 3], [30, 5], [31, 7], [33, 4], [33, 1], [40, 3], [40, 0], [40, 2],
↪ [41, 4], [44, 1], [50, 3], [50, 0], [50, 2]]
getQ: [[0, 0], [33, 1], [40, 0], [44, 1], [50, 0]]
putQ [[0, 0], [0, 1], [0, 2], [0, 3], [0, 2], [0, 1], [0, 0], [10, 1], [11, 0]]
```

[Return to *Top*]

Reneging

The `yield put` can be subject to *reneging* using one of the compound statements:

- `yield (put,self,sObj,give [,P]),(hold,self,waittime)`

where if the process cannot put the list of objects in *give* before *waittime* is elapsed, the offerer leaves the `putQ` and its execution continues or

- `yield (put,self,sObj,give [,P]),(waitevent,self,events)`

where if one of the SimEvents in *events* occurs before it can put the list of objects in *give* the offerer leaves the `putQ` and its execution continues.

In either case if reneging has *not* occurred the list of objects in *give* will have been put into the Store and `self.stored(Sobj)` will be `True`.

The mandatory pattern for a `put` with reneging is:

```
yield (put, self, sObj, give [,P]),(<reneging clause>)
if self.stored(sObj):
   ## process  did not renege
   . . . .
else:
   ## process reneged before being able to put into the resource
```

This is so because `self.stored( )` not only tests for reneging, but it also cleanly removes a reneging process from the `putQ`.

The `yield get` can be subject to similar *reneging* using one of the compound statements:

- `yield (get, self, sObj, n [,P]), (hold, self, waittime)`
- `yield (get, self, sObj, ffn [,P]), (hold, self, waittime)`

where if the process does not acquire the amount before *waittime* is elapsed, the offerer leaves the `waitQ` and its execution continues.

- `yield (get, self, sObj, n [,P]), (waitevent, self, events)`
- `yield (get, self, sObj, ffn [,P]), (waitevent, self, events)`

where if one of the SimEvents in *events* occurs before enough becomes available, reneging occurs, the offerer leaves the `waitQ` and its execution continues.

In either case if reneging has *not* occurred `self.got` contains the list of retrieved objects and `self.acquired(Sobj)` will be `True`.

The mandatory pattern for a `get` with reneging is:

```
yield (get, self, lev, sObj, <n or ffn> [,P]),(<reneging clause>)
if self.acquired(sObj):
   ## process  did not renege,
   . . . .
else:
   ## process reneged before being able to put into the resource
```

This is so because `self.acquired( )` not only tests for reneging, but it also cleanly removes a reneging process from the `getQ`.

[Return to *Top*]

Storing objects in an order

The contents of a Store instance are listed in a queue. By default, this list is kept in FIFO order. However, the list can be kept in a user-defined order. You do this by defining a function for reordering the list and adding it to the Store instance for which you want to change the list order. Subsequently, the SimPy system will automatically call that function after any addition (`put`) to the queue.

Example

```
class Parcel:
     def __init__(self, weight):
         self.weight = weight

lightFirst=Store()

def getLightFirst(self, par):
     """Lighter parcels to front of queue"""
     tmplist = [(x.weight, x) for x in par]
     tmplist.sort()
     return [x for (key, x) in tmplist]

lightFirst.addSort(getLightFirst)
```

Now any `yield get` will get the lightest parcel in `lightFirst`'s queue.

The `par` parameter is automatically given the Store's buffer list as value when the SimPy run time system calls the re-ordering function.

<aStore>.addSort(<reorderFunction>) adds a re-order function to <aStore>.

Note that such function only changes the sorting order of the Store instance, NOT of the Store class.

Master/Slave modelling with a Store

The items in a `Store` can be of any Python type. In particular, they may be SimPy processes. This can be used to model a Master/Slave situation – an asymmetrical cooperation between two or more processes, with one process (the Master) being in charge of the cooperation.

The consumer (Master) requests one or more Slaves to be added to the Store's contents by the Producer (which may be the same process as the Slave). For Master/Slave cooperation, the Slave has to be passivated (by a `yield passivate` or `yield waitevent` statement) after it is `put` and reactivated when it is retrieved and finished with. As this is NOT done automatically by the `Store`, the Master has to signal the end of the cooperation. This Master/Slave pattern results in the slave process' life-cycle having a hole between the slave process arrival and its departure after having been served.

Example Cars arrive randomly at a car wash and add themselves to the `waitingCars` queue. They wait (passively) for a `doneSignal`. There are two `Carwash` washers. These `get` a car, if one is available, wash it, and then send the `doneSignal` to reactivate it. We elect to model the `Carwash` as Master and the `Cars` as slaves.

Four cars are put into the `waiting` list and these make up the initial set of cars waiting for service. Additional cars are generated randomly by the `CarGenerator` process. Each car `yield puts` itself onto the `waitingCars Store` and immediately passivates itself by waiting for a `doneSignal` from a car washer. The car washers cycle round `getting` the next car on the queue, washing it and then sending a `doneSignal` to it when it has finished:

```
from SimPy.Simulation import (Process, SimEvent, Store, activate, get,
                              initialize, hold, now, put, simulate, waitevent)
"""Carwash is master"""


class Carwash(Process):
    """Carwash is master"""

    def __init__(self, name):
        Process.__init__(self, name=name)

    def lifecycle(self):
        while True:
            yield get, self, waitingCars, 1
            carBeingWashed = self.got[0]
            yield hold, self, washtime
            carBeingWashed.doneSignal.signal(self.name)


class Car(Process):
    """Car is slave"""

    def __init__(self, name):
        Process.__init__(self, name=name)
        self.doneSignal = SimEvent()

    def lifecycle(self):
        yield put, self, waitingCars, [self]
        yield waitevent, self, self.doneSignal
        whichWash = self.doneSignal.signalparam
        print('%s car %s done by %s' % (now(), self.name, whichWash))


class CarGenerator(Process):
    def generate(self):
        i = 0
        while True:
            yield hold, self, 2
            c = Car('%d' % i)
            activate(c, c.lifecycle())
            i += 1


washtime = 5
initialize()

# put four cars into the queue of waiting cars
for j in range(1, 5):
    c = Car(name='%d' % -j)
    activate(c, c.lifecycle())

waitingCars = Store(capacity=40)
for i in range(2):
    cw = Carwash('Carwash %s' % i)
    activate(cw, cw.lifecycle())

cg = CarGenerator()
activate(cg, cg.generate())
```

```
simulate(until=30)
print('waitingCars %s' % [x.name for x in waitingCars.theBuffer])
```

The output of this program, running to time 30, is:

```
5 car -1 done by Carwash 0
5 car -2 done by Carwash 1
10 car -3 done by Carwash 0
10 car -4 done by Carwash 1
15 car 0 done by Carwash 0
15 car 1 done by Carwash 1
20 car 2 done by Carwash 0
20 car 3 done by Carwash 1
25 car 4 done by Carwash 0
25 car 5 done by Carwash 1
30 car 6 done by Carwash 0
30 car 7 done by Carwash 1
waitingCars ['10', '11', '12', '13', '14']
```

It is also possible to model this car wash with the cars as Master and the `Carwash` as Slaves.

[Return to *Top*]

2.1.7 Random Number Generation

Simulations usually need random numbers. As SimPy does not supply random number generators of its own, users need to import them from some other source. Perhaps the most convenient source is the standard Python random module. It can generate random variates from the following continuous distributions: uniform, beta, exponential, gamma, normal, log-normal, weibull, and vonMises. It can also generate random variates from some discrete distributions. Consult the module's documentation for details. (Excellent brief descriptions of these distributions, and many others, can be found in the Wikipedia.)

Python's `random` module can be used in two ways: you can import the methods directly or you can import the `Random` class and make your own random objects. In the second method, each object gives a different random number sequence, thus providing multiple random streams as in Simscript and ModSim.

Here the first method is described (and minimally at that). A single pseudo-random sequence is used for all calls. You `import` the methods you need from the `random` module. For example:

```
from random import seed, random, expovariate, normalvariate
```

In simulation it is good practice to set the initial `seed` for the pseudo-random sequence at the start of each run. Then you have control over the random numbers used. Replications and comparisons are easier and, together with variance reduction techniques, can provide more accurate estimates. In the following code snippet we set the initial seed to 333555. `X` and `Y` are pseudo-random variates from the two distributions. Both distributions have the same mean:

```
1 from random import seed, expovariate, normalvariate
2
3 seed(333555)
4 X = expovariate(0.1)
5 Y = normalvariate(10.0, 1.0)
```

[Return to *Top*]

2.1.8 Recording Simulation Results

The `Tally` and `Monitor` class objects enable us to observe a single variable of interest and to return a simple data summary either during or at the completion of a simulation run.

Both use the `observe` method to record data on one variable. For example we might use a Monitor object to record the waiting times for a sequence of customers and another to record the total number of customers in the shop. In a discrete-event system the number of customers changes only at arrival or departure events and it is at those events that the waiting times and number in the shop must be observed. Monitors and Tallys provide elementary statistics useful either alone or as the start of a more sophisticated statistical analysis and have proved invaluable in many simulations.

A few more tools associated with recording results are:

- All Monitors are registered automatically in the global list variable `allMonitors` and all Tallys in variable `allTallies`. When a simulation is completed results can easily be tabulated and summarized using these lists.
- The function `startCollection()` can be called to initialize Monitors and Tallys at a certain simulation time. This is helpful when a simulation needs a 'warmup' period to achieve steady state before measurements are started.

Defining Tallys and Monitors

The ''Tally" class records enough information (such as sums and sums of squares) while the simulation runs to return simple data summaries. This has the advantage of speed and low memory use. Tallys can also furnish data for a histogram. However, they do not preserve a time-series usable in more advanced statistical analysis. When a Tally is defined it is automatically added to the global list `allTallies`.

To define a new Tally object:

- `m = Tally(name='a_Tally', ylab='y', tlab='t')`
- `name` is a descriptive name for the tally object (default='`a_Tally`').
- `ylab` and `tlab` are descriptive labels used by the `SimPy.SimPlot` package when plotting graphs of the recorded data. They default to `'y'` and `'t'`, respectively. (If a *histogram* is required the method `setHistogram` must be called before recording starts).

The `Monitor` class preserves a complete time-series of the observed data values, *y*, and their associated times, *t*. It calculates the data summaries using these series only when they are needed. It is slower and uses more memory than `Tally`. In long simulations its memory demands may be a disadvantage. When a Monitor is defined it is automatically added to the global list `allMonitors`.

To define a new Monitor object:

- `m = Monitor(name='a_Monitor', ylab='y', tlab='t')`
- `name` is a descriptive name for the Monitor object (default='`a_Monitor`').
- `ylab` and `tlab` are descriptive labels used by the `SimPy.SimPlot` package when plotting graphs of the recorded data. They default to `'y'` and `'t'`, respectively. (A *histogram* can be requested at any time).

Observing data

Both Tallys and Monitors use the `observe` method to record data. Here and in the next section, *r* is either a Tally or a Monitor object:

- `r.observe(y [,t])` records the current value of the variable, `y` and time *t* (or the current time, `now( )`, if *t* is missing). A Monitor retains the two values as a sub-list `[t,y]`. A Tally uses them to update the accumulated statistics.

To assure that time averages are calculated correctly `observe` should be called immediately *after* a change in the variable. For example, if we are using Monitor *r* to record the number *N* of jobs in a system, the correct sequence of commands on an arrival is:

```
N = N + 1     # FIRST, increment the number of jobs
r.observe(N)  # THEN observe the new value of N using r
```

The recording of data can be `reset` to start at any time in the simulation:

- `r.reset([t])` resets the observations. The recorded data is re-initialized, and the observation starting time is set to *t*, or to the current simulation time, `now( )`, if *t* is missing.

Data summaries

The following simple data summaries can be obtained from either Monitors or Tallys at any time during or after the simulation run:

- `r.count( )`, the current number of observations. (If *r* is a Monitor this is the same as `len(r)`).
- `r.total( )`, the sum of the `y` values
- `r.mean( )`, the simple average of the observed `y` values, ignoring the times at which they were made. This is `r.total( )/N` where `N=r.count( )`. (If there are no observations, the message: "SimPy: No observations for mean" is printed). See *Recording Resource queue lengths* for the difference between the simple or numerical average and the time-average.

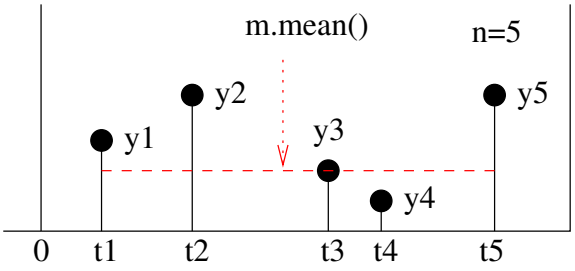


Fig. 2.1: `r.mean` is the simple average of the `y` values observed.

- `r.var( )` the *sample* variance of the observations, ignoring the times at which they were made. If an unbiased estimate of the *population* variance is desired, the sample variance should be multiplied by *n/(n-1)*, where *n* = *r.count( )*. In either case the standard deviation is, of course, the square-root of the variance (If there are no observations, the message: "SimPy: No observations for sample variance" is printed).
- `r.timeAverage([t])` the time-weighted average of `y`, calculated from time 0 (or the last time `r.reset([t])` was called) to time *t* (or to the current simulation time, `now( )`, if *t* is missing). This is determined from the area under the graph shown in the figure, divided by the total time of observation. For accurate time-average results `y` most be piecewise constant and `observed` just after each change in its value. (If there are no observations, the message "SimPy: No observations for timeAverage" is printed. If no time has elapsed, the message "SimPy: No elapsed time for timeAverage" is printed).
- `r.timeVariance([t])` the time-weighted variance of the `y` values calculated from time 0 (or the last time `r.reset([t])` was called) to time *t* (or to the current simulation time, `now()`, if *t* is missing).
- `r.__str__( )` is a string that briefly describes the current state of the monitor. This can be used in a print statement.

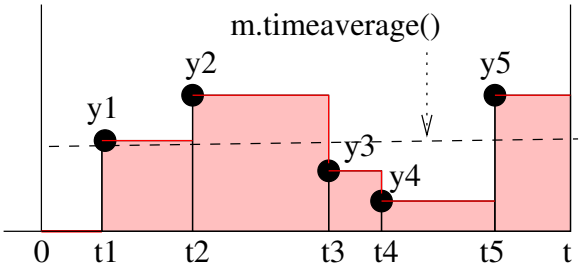


Fig. 2.2: `r.timeAverage( )` is the time-weighted average of the observed `y` values. Each `y` value is weighted by the time for which it exists. The average is the area under the above curve divided by the total time, *t*.

Special methods for Monitor

The `Monitor` variety of Recorder is a sub-class of `List` and has a few extra methods:

- `m[i]` holds the observation `i` as a two-item list, *[ti, yi]*
- `m.yseries( )` is a list of the recorded data values, *yi*
- `m.tseries( )` is a list of the recorded times, *ti*

Histograms

A `Histogram` is a derived class of `list` that counts the observations that fall into a number of specified ranges, called bins. A histogram object can be displayed either by printing it out in text form using `printHistogram` method or using the `plotHistogram` method in the `SimPy.SimPlot` package.

- `h = Histogram(low=<float>, high=<float>, nbins=<integer>)` is a histogram object that counts the number of `y` values in each of its bins, based on the recorded `y` values.
  - `low` is the nominal lowest value of the histogram (default=0.0)
  - `high` is the nominal highest value of the histogram (default=100.0)
  - `nbins` is the number of bins between `low` and `high` into which the histogram is to be divided (default=10). SimPy automatically constructs an additional two bins to count the number of `y` values `under` the `low` value and the number `over` the `high` value. Thus, the total number of bins actually used is `nbins + 2`. The number of `y` values in each of these bins is counted and assigned to the appropriate bin.

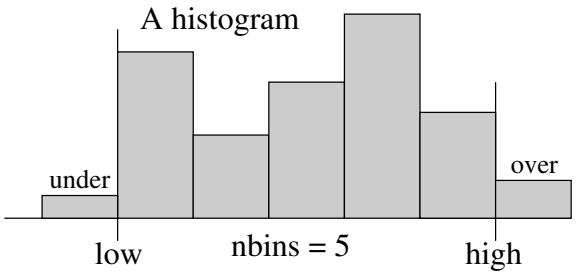


Fig. 2.3: A Histogram contains the number of observed `y` values falling into each of its `nbins+2` bins.

A Histogram, *h*, can be printed out in text form using

- `h.printHistogram(fmt="%s")` prints out a histogram in a standard format.
  - `fmt` is a python string format for the bin range values.

Example Printing a histogram from a Tally:

```
from SimPy.Simulation import Tally
import random as r

t = Tally(name="myTally", ylab="wait time (sec)")
t.setHistogram(low=0.0, high=1.0, nbins=10)
for i in range(100000):
    t.observe(y=r.random())
print(t.printHistogram(fmt="%6.4f"))
```

This gives a printed histogram like this:

```
Histogram for myTally:
Number of observations: 100000
          wait time (sec) < 0.0000:      0 (cum:      0/  0.0%)
0.0000 <= wait time (sec) < 0.1000:  10135 (cum:  10135/ 10.1%)
0.1000 <= wait time (sec) < 0.2000:   9973 (cum:  20108/ 20.1%)
0.2000 <= wait time (sec) < 0.3000:  10169 (cum:  30277/ 30.3%)
0.3000 <= wait time (sec) < 0.4000:  10020 (cum:  40297/ 40.3%)
0.4000 <= wait time (sec) < 0.5000:  10126 (cum:  50423/ 50.4%)
0.5000 <= wait time (sec) < 0.6000:   9866 (cum:  60289/ 60.3%)
0.6000 <= wait time (sec) < 0.7000:   9910 (cum:  70199/ 70.2%)
0.7000 <= wait time (sec) < 0.8000:   9990 (cum:  80189/ 80.2%)
0.8000 <= wait time (sec) < 0.9000:   9852 (cum:  90041/ 90.0%)
0.9000 <= wait time (sec) < 1.0000:   9959 (cum: 100000/100.0%)
1.0000 <= wait time (sec)          :      0 (cum: 100000/100.0%)
```

Although both Tallys and Monitors can return a histogram of the data, they furnish histogram data in different ways.

- The Tally object accumulates the histogram's bin counts as each value is observed during the simulation run. Since none of the individual values are preserved, the `setHistogram` method must be called to provide a histogram object to hold the accumulated bin counts before any values are actually observed.
- The Monitor object stores all its data, so the accumulated bin counts can be computed whenever they are desired. Thus, the histogram need not be set up until it is needed and this can be done after the data has been gathered.

Setting up a Histogram for a Tally object

To establish a histogram for a Tally object, `r`, we call the `setHistogram` method with appropriate arguments before we observe any data, e.g.,

- `r.setHistogram(name = '', low=0.0, high=100.0, nbins=10)`

As usual, `name` is a descriptive title for the histogram (defaults to blank). Then, after `observing` the data:

- `h = r.getHistogram( )` returns a completed histogram using the histogram parameters as set up.

Example In the following example we establish a `Tally` recorder to observe values of an exponential random variate. It uses a histogram with 30 bins (plus the under- and over-count bins):

```
from SimPy.Simulation import Tally
from random import expovariate

r = Tally('Tally')  # define a tally object, r
r.setHistogram(name='exponential',
               low=0.0, high=20.0, nbins=30)  # set before observations

for i in range(1000):    # make the observations
    y = expovariate(0.1)
    r.observe(y)

h = r.getHistogram()     # return the completed histogram
print(h)
```

Setting up a Histogram for a Monitor object

For Monitor objects, a histogram can be set up and returned in a single call, e.g.,

- `h = r.histogram(low=0.0,high=100.0,nbins=10)`

This call is equivalent to the following pair:

- `r.setHistogram(name = '', low=0.0, high=100.0, nbins=10)`
- `h = r.getHistogram( )`, which returns the completed histogram.

Example Here we establish a `Monitor` to observe values of an exponential random variate. It uses a histogram with 30 bins (plus the under- and over-count bins):

```
from SimPy.Simulation import Monitor
from random import expovariate

m = Monitor()        # define the Monitor object, m

for i in range(1000):    # make the observations
    y = expovariate(0.1)
    m.observe(y)

# set up and return the completed histogram
h = m.histogram(low=0.0, high=20, nbins=30)
```

[Return to *Top*]

2.1.9 Other Links

Several example SimPy models are included with the SimPy code distribution in the file `SimPyModels`.

Klaus Muller and Tony Vignaux, *SimPy: Simulating Systems in Python*, O'Reilly ONLamp.com, 2003-Feb-27, http://archive.oreilly.com/pub/a/python/2003/02/27/simpy.html

Norman Matloff, *Introduction to the SimPy Discrete-Event Simulation Package*, U Cal: Davis, 2003, http://heather.cs.ucdavis.edu/~matloff/simcourse.html

David Mertz, *Charming Python: SimPy simplifies complex models*, IBM Developer Works, Dec 2002, https://www.ibm.com/developerworks/library/l-simpy/index.html

[Return to *Top*]

2.1.10 Acknowledgments

We thank those users who have sent comments to correct or improve this text. These include: F. Benichu, Bob Helmbold, M. Matti. We will be grateful for further corrections or suggestions.

2.1.11 Appendices

A0. Changes from the previous version of SimPy

SimPy 2.2b1 differs from version 2.1 in the following ways:

Additions:

Changes:

- The Unit tests have been rewritten
- The directory structure of the release has been simplified
- The documentation has had some minor changes

A1. SimPy Error Messages

Advisory messages

These messages are returned by `simulate( )`, as in `message=simulate(until=123)`.

Upon a normal end of a simulation, `simulate( )` returns the message:

- **SimPy: Normal exit**. This means that no errors have occurred and the simulation has run to the time specified by the `until` parameter.

The following messages, returned by `simulate( )`, are produced at a premature termination of the simulation but allow continuation of the program.

- **SimPy: No more events at time x**. All processes were completed prior to the *endtime* given in *simulate(until=endtime)*.
- **SimPy: No activities scheduled**. No activities were scheduled when *simulate( )* was called.

Fatal error messages

These messages are generated when SimPy-related fatal exceptions occur. They end the SimPy program. Fatal SimPy error messages are output to *sysout*.

- **Fatal SimPy error: activating function which is not a generator (contains no 'yield')**. A process tried to (re)activate a function which is not a SimPy process (=Python generator). SimPy processes must contain at least one *yield ...* statement.
- **Fatal SimPy error: Simulation not initialized**. The SimPy program called *simulate( )* before calling *initialize( )*.

- **SimPy: Attempt to schedule event in the past**: A *yield hold* statement has a negative delay time parameter.
- **SimPy: initialBuffered exceeds capacity**: Attempt to initialize a Store or Level with more units in the buffer than its capacity allows.
- **SimPy: initialBuffered param of Level negative: x**: Attempt to initialize a Level with a negative amount x in the buffer.
- **SimPy: Level: wrong type of initialBuffered (parameter=x)**: Attempt to initialize a buffer with a non-numerical initial buffer content x.
- **SimPy: Level: put parameter not a number**: Attempt to add a non-numerical amount to a Level's buffer.
- **SimPy: Level: put parameter not positive number**: Attempt to add a negative number to a Level's amount.
- **SimPy: Level: get parameter not positive number: x**: Attempt to get a negative amount x from a Level.
- **SimPy: Store: initialBuffered not a list**: Attempt to initialize a Store with other than a list of items in the buffer.
- **SimPy: Item to put missing in yield put stmt**: A *yield put* was malformed by not having a parameter for the item(s) to put into the Store.
- **SimPy: put parameter is not a list**: *yield put* for a Store must have a parameter which is a list of items to put into the buffer.
- **SimPy: Store: get parameter not positive number: x**: A *yield get* for a Store had a negative value for the number to get from the buffer.
- **SimPy: Fatal error: illegal command: yield x**: A *yield* statement with an undefined command code (first parameter) x was executed.

Monitor error messages

- **SimPy: No observations for mean**. No observations were made by the monitor before attempting to calculate the mean.
- **SimPy: No observations for sample variance**. No observations were made by the monitor before attempting to calculate the sample variance.
- **SimPy: No observations for timeAverage**, No observations were made by the monitor before attempting to calculate the time-average.
- **SimPy: No elapsed time for timeAverage**. No simulation time has elapsed before attempting to calculate the time-average.

A2. SimPy Process States

From the viewpoint of the model builder a SimPy process, *p*, can at any time be in one of the following states:

- **Active**: Waiting for a scheduled event. This state simulates an activity in the model. Simulated time passes in this state. The process state *p.active( )* returns *True*.
- **Passive**: Not active or terminated. Awaiting *(re-)activation* by another process. This state simulates a real world process which has not finished and is waiting for some trigger to continue. Does not change simulation time. *p.passive( )* returns *True*.
- **Terminated**: The process has executed all its action statements. If referenced, it serves as a data instance. *p.terminated( )* returns *True*

Initially (upon creation of the Process instance), a process returns *passive*.

In addition, a SimPy process, *p*, can be in the following (sub)states:

- **Interrupted**: Active process has been interrupted by another process. It can immediately respond to the interrupt. This simulates an interruption of a simulated activity before its scheduled completion time. *p.interrupted( )* returns *True*.
- **Queuing**: Active process has requested a busy resource and is waiting (passive) to be reactivated upon resource availability. *p.queuing(a_resource)* returns *True*.

A3. SimPlot, The SimPy plotting utility

`SimPlot` provides an easy way to graph the results of simulation runs.

A4. SimGUI, The SimPy Graphical User Interface

`SimGUI` provides a way for users to interact with a SimPy program, changing its parameters and examining the output.

A5. SimulationTrace, the SimPy tracing utility

`SimulationTrace` has been developed to give users insight into the dynamics of the execution of SimPy simulation programs. It can help developers with testing and users with explaining SimPy models to themselves and others (e.g., for documentation or teaching purposes).

A6. SimulationStep, the SimPy event stepping utility

`SimulationStep` can assist with debugging models, interacting with them on an event-by-event basis, getting event-by-event output from a model (e.g. for plotting purposes), etc.

It caters for:

- running a simulation model, while calling a user-defined procedure after every event,
- running a simulation model one event at a time by repeated calls,
- starting and stopping the event-stepping mode under program control.

A7. SimulationRT, a real-time synchronizing utility

`SimulationRT` allows synchronizing simulation time and real (wall-clock) time. This capability can be used to implement, e.g., interactive game applications or to demonstrate a model's execution in real time.

[Return to *Top*]

2.1.12 Glossary

(Note: Terms in *italics* refer to other special terms. Items in `code font` are code fragments or specific code names.):

activeQ A *Resource* object automatically creates and maintains its own activeQ, the queue (list) of process objects that are currently using one of the Resource's units. See *Resources*. (See also the Glossary entry for *waitQ*.)

activate Commands a *process object* to being executing its *PEM*. See *Starting and stopping SimPy process objects*.

Backus-Naur Form (BNF) notation This manual occasionally uses a modified Backus-Naur Form notation to exhibit command syntax, as in the description of the *activate* command:

```
activate(p, p.PEM([args]) [,{at=t|delay=period}] [,prior=False])
```

In this notation, square brackets [ ] indicate items that are optional, braces { } indicate items of which zero or more may be present, and a vertical bar | indicates a choice between alternatives (with none of them being a possibility).

cancel Deletes all of a *process object's* scheduled future events. See *Starting and stopping SimPy process objects*.

entity An alternative name for *process object*.

event A SimEvent object. See *Advanced synchronization/scheduling capabilities*.

FIFO An attribute of a resource object (i.e., a *Resource*, *Level*, or *Store*) indicating that an associated queue (e.g., the *ActiveQ*, *waitQ*, *getQ*, or *putQ*) is to be kept in FIFO order. (See also the Glossary entries for *PriorityQ* and *qType*.)

getQ The queue of processes waiting to take something from a *Level* or *Store* resource. See also the Glossary entry for *putQ*.

interrupt Requests a "victim" *process object* to interrupt (i.e., to immediately and prematurely end) its current `yield hold,...` command. (Note: A process object cannot interrupt itself.) See *Asynchronous interruptions*.

Level A particular type of *resource facility* that models the production and consumption of a homogeneous undifferentiated "material." *Process objects* can increase or decrease the amount of material in a Level resource facility. See *Levels*.

Monitor A data recorder that compiles basic statistics as a function of time on variables such as waiting times and queue lengths. (Note: Monitors can also preserve complete time-series data for post-simulation analyses.) See *Recording Simulation Results*.

monitorType The type of *Recorder* to be used for recording simulation results. Usually this is either a *Monitor* or a *Tally*. (See also the Glossary entry for *Recorder*.)

monitored A (boolean) attribute of a *resource* object indicating whether to keep a record of its activity. See *Recorder*.

passivate Halts ("freezes") a *process object's* PEM. The process object becomes "passive". See *Starting and stopping SimPy Process Objects*.

PEM An abbreviation for *Process Execution Method*, q.v.

preempt To force a *process* object currently using a *resource* unit to release it and make it available for use by another process object. See *Preemptive requests for a Resource unit*.

preemptable A settable attribute of *Resource* objects. The Resource object's units are preemptable if `preemptable==True`, otherwise not. See *Preemptive requests for a Resource unit*.

priority A non-negative integer or real value controlling the order of *process* objects in a queue. Higher values represent higher priority. Higher priority process objects are placed ahead of lower priority ones in the queue. See also the Glossary entry for *FIFO*.

PriorityQ An attribute of a resource object (i.e., a *Resource*, *Level*, or *Store*) indicating that an associated queue (e.g., the *ActiveQ*, *waitQ*, *getQ*, or *putQ*) is to be kept in order of *priority*. (See also the Glossary entries for *FIFO*, *qType*.)

process We usually call both process objects and their classes "processes" (with a small "p"). Thus, "process" may refer to a *Process class* or to a *process object*, depending on context. To avoid ambiguity or for added emphasis we often explicitly state whether a class or an object is intended.

Process class A class that inherits from SimPy's `Process` class and contains at least one *Process Execution Method*. Process classes may also contain other methods – in particular they may contain an `__init__` method. See *Processes*.

Process Execution Method A *Process class* method that contains at least one `yield ...` statement. See *Defining a process*.

process object An object created from (i.e., an instance of) a *Process class*. See *Processes*.

putQ The queue of processes waiting to add something to a *Level* or *Store* resource. See also the Glossary entry for *getQ*.

reactivate Reactivates ("unfreezes") a passivated or a terminated *process object's* PEM. The *process object* becomes "active". See *Starting and stopping SimPy Process Objects*.

Recorder A device for recording simulation results. Unless otherwise specified, it usually refers either to a *Monitor* or a *Tally*. However, Recorders also include histograms and observers. See *Recording Simulation Results* for *Monitors*, *Tallys*, and the other devices for recording simulation results.

renege To leave a queue before acquiring a resource unit. See *Reneging – leaving a queue before acquiring a resource*.

resource Same as "resource facility." A congestion point at which *process objects* may need to queue for access to resources. The term "resource" (with a small "r") is used as a generic term for the individual resource facilities provided by SimPy (i.e., *Resources*, *Levels*, and *Stores*).

qType An attribute of *resource* objects indicating whether an associated queue is to be kept in *FIFO* or *PriorityQ* order. See the Glossary entries for *waitQ*, *ActiveQ*, *putQ*,and *getQ*. See also the treatment of these queues in the sections on the individual resources (i.e., *Resources*, *Levels*, and *Stores*).

Resource A particular type of *resource facility* that possesses several identical *resource units*. A *process object* may acquire one (and only one) of the Resource's resource units. See *Resources* .

Resource unit One of the individual resources associated with a *Resource* type of *resource facility*. See *Resources*.

SimEvent The SimPy class for defining and creating SimEvent objects. Occasionally designates a SimEvent object when context makes that usage clear. See *Advanced synchronization/scheduling capabilities*.

Store A particular type of *resource facility* that models the production and consumption of individual items. *Process objects* can insert or remove items from the Store's list of available items. See *Stores*.

Tally A particular type of *Recorder* that compiles basic statistics as a function of time on variables such as waiting times and queue lengths. (Note: Tallys do not preserve complete time-series data for post-simulation analyses.) See *Recording Simulation Results*. (See also the Glossary entry for *monitorType*.)

unit (of a Resource) One of the individual resource capabilities provided by a *Resource*. See *Resources*.

waitQ A *Resource* object automatically creates and maintains its own *waitQ*, the queue (list) of process objects that have requested but not yet received one of the Resource's units. See *Resources*. (See also the Glossary entry for *activeQ*.)

2.2 SimPy Classic's Object Oriented API

Authors

- Klaus Muller <Muller@users.sourceforge.net>

Release 2.3.3

Python Version 2.7 and later

Date Feb 24, 2018

Contents

2.2.1 Introduction

This document describes the object oriented (OO) programming interface introduced with SimPy 2.0. This is an add-on to the existing API, an alternative API. There is full backward compatibility:

Motivation

Many simulation languages support a procedural modelling style. Using them, problems are decomposed into procedures (functions, subroutines) and either represented by general components, such as queues, or represented in code with data structures.

There are fundamental problems with using the procedural style of modelling and simulation. Procedures do not correspond to real world components. Instead, they correspond to methods and algorithms. Mapping from the real (problem) world to the model and back is difficult and not obvious, particularly for users expert in the problem domain, but not in computer science. Perhaps the greatest limitation of the procedural style is the lack of model extensibility. The only way in this style to change simulation models is through functional extension. One can add structural functionality but not alter any of its basic processes.

Right from its beginning, SimPy, on the other hand, has supported an **object oriented approach** to simulation modelling. In SimPy, models can be implemented as collections of autonomous, cooperating objects. These objects are self-sufficient and independent. The actions on these objects are tied to the objects and their attributes. The object-oriented capabilities of Python strongly support this encapsulation.

Why does this matter for simulation models? It helps with the mapping from real-world objects and their activities to modelled objects and activities, and back. This not only reduces the complexity of the models, it also makes for easier validation of models and interpretation of simulation results in real world terms.

The new API allows different, often more concise, cleaner program patterns. It strongly supports the development of libraries of model components for specific real world domains. It also supports the re-use and extension of models when model specifications change. In particular larger SimPy programs written with the advanced OO API should be easier to maintain and extend. Users are advised to familiarize themselves with this programming paradigm by reading the models in the *SimPyModels* folder. Most of them are provided in two implementations, i.e. in the existing and in the OO API. Similarly, the programs in the Bank tutorials are provided with both APIs.

The advanced OO API has been developed very elegantly by Stefan Scherfke and Ontje Lünsdorf, starting from SimPy 1.9. Thanks, guys, for this great job!

Readers of this document should be familiar with the basics of SimPy and have read at least "Basic SimPy - Manual For First Time Users". They should also know how subclassing is done in Python.

2.2.2 Basic SimPy OO API Design

A class `Simulation` has been added to module `SimPy.Simulation`. `SimulationTrace`, `SimulationStep` and `SimulationRT` are subclasses of `Simulation`. Multiple instances of these classes can

co-exist in a SimPy program.

Backward compatibility

Since SimPy 2.0, the package offers both the existing procedural API and an object-oriented API where simulation capabilities are provided by instantiating `Simulation`. `SimulationTrace`, `SimulationStep` or `SimulationRT` are subclasses of `Simulation`.

Each `SimulationXX` instance has its own event list and therefore its own simulation time. A `SimulationXX` instance can effectively be considered as a simulated, isolated parallel world. Any *Process*, *Resource*, *Store*, *Level*, *Monitor*, *Tally* or *SimEvent* instance belongs to one and only one world (i.e., `Simulationxx` instance).

The following program shows what this means for API and program structure:

```
from SimPy.Simulation import (Simulation, Process, Resource, request, hold,
                              release)
"""Object Oriented SimPy API"""

# Model components ------------------------------


class Car(Process):
    def run(self, res):
        yield request, self, res
        yield hold, self, 10
        yield release, self, res
        print("Time: %s" % self.sim.now())

# Model and Experiment --------------------------


s = Simulation()
s.initialize()
r = Resource(capacity=5, sim=s)
auto = Car(sim=s)
s.activate(auto, auto.run(res=r))
s.simulate(until=100)
```

Using the existing API, the following program is semantically the same and also works under the OO version:

```
from SimPy.Simulation import (activate, hold, initialize, now, request,
                              release, simulate, Process, Resource)
"""Traditional SimPy API"""

# Model components ------------------------------


class Car(Process):
    def run(self, res):
        yield request, self, res
        yield hold, self, 10
        yield release, self, res
        print("Time: %s" % now())

# Model and Experiment --------------------------


initialize()
```

```
r = Resource(capacity=5)
auto = Car()
activate(auto, auto.run(res=r))
simulate(until=100)
```

This full (backwards) compatibility is achieved by the automatic generation of a *SimulationXX* instance "behind the scenes".

Models as SimulationXX subclasses

The advanced OO API can be used to generate model classes which are SimulationXX subclasses. This ties a model and a SimulationXX instance together beautifully. See the following example:

```
# CarModel.py
import SimPy.Simulation as simulation
"""Advanced Object Oriented SimPy API"""

# Model components ------------------------------


class Car(simulation.Process):
    def park(self):
        yield simulation.request, self, self.sim.parking
        yield simulation.hold, self, 10
        yield simulation.release, self, self.sim.parking
        print("%s done at %s" % (self.name, self.sim.now()))

# Model ----------------------------------------


class Model(simulation.Simulation):
    def __init__(self, name, nrCars, spaces):
        simulation.Simulation.__init__(self)
        self.name = name
        self.nrCars = nrCars
        self.spaces = spaces

    def runModel(self):
        # Initialize Simulation instance
        self.initialize()
        self.parking = simulation.Resource(name="Parking lot",
                                           unitName="spaces",
                                           capacity=self.spaces, sim=self)
        for i in range(self.nrCars):
            auto = Car(name="Car%s" % i, sim=self)
            self.activate(auto, auto.park())
        self.simulate(until=100)


if __name__ == "__main__":

    # Experiment ------------------------------------
    myModel = Model(name="Experiment 1", nrCars=10, spaces=5)
    myModel.runModel()
    print(myModel.now())
```

class `Model` here is a subclass of `Simulation`. Every model execution, i.e. call to `runModel`, reinitializes the

simulation (creates an empty event list and sets the time to 0) (see line 24). `runModel` can thus be called repeatedly for multiple runs of the same experiment setup:

```
if __name__=="__main__":

    ## Experiments ---------------------------------

    myModel = Model(name="Experiment 1", nrCars=10, spaces=5)
    for repetition in range(100):

    ## One Experiment ------------------------------

        myModel.runModel()
        print(myModel.now())
```

Model extension by subclassing

With the advanced OO API, it is now very easy and clean to extend a model by subclassing. This effectively allows the creation of model libraries.

For example, the model in the previous example can be extended to one in which also vans compete for parking spaces. This is done by importing the `CarModel` module and subclassing `Model` as follows:

```
# CarModelExtension.py

# Model components ----------------------------

import SimPy.Simulation as simulation
import CarModel


class Van(simulation.Process):
    def park(self):
        yield simulation.request, self, self.sim.parking
        yield simulation.hold, self, 5
        yield simulation.release, self, self.sim.parking
        print("%s done at %s" % (self.name, self.sim.now()))


# Model ---------------------------------------


class ModelExtension(CarModel.Model):
    def __init__(self, name, nrCars, spaces, nrTrucks):
        CarModel.Model.__init__(self, name=name, nrCars=nrCars, spaces=spaces)
        self.nrTrucks = nrTrucks

    def runModel(self):
        self.initialize()
        self.parking = simulation.Resource(name="Parking lot",
                                           capacity=self.spaces,
                                           sim=self)
        for i in range(self.nrCars):
            auto = CarModel.Car(name="Car%s" % i, sim=self)
            self.activate(auto, auto.park())
        for i in range(self.nrTrucks):
            truck = Van(name="Van%s" % i, sim=self)
```

```
            self.activate(truck, truck.park())
        self.simulate(until=100)


# Experiment ----------------------------------

myModel1 = ModelExtension(name="Experiment 2", nrCars=10, spaces=5, nrTrucks=3)
myModel1.runModel()
```

Let's walk through this:

***Lines 9-14*:** Addition of a `Van` class with a `park` PEM.

***Line 20*:** Definition of a subclass `ModelExtension` which extends class `Model`.

***Lines 22-23*:** Initialization of the model class (`Model`) from which `ModelExtension` is derived. When subclassing a class in Python, this is always necessary: Python does **not** automatically initialize the super-class.

***Lines 25-36*:** Defines a `runModel` method for `ModelExtension` which also generates and activates `Van` objects.

2.2.3 API changes

Module `SimPy.Simulation`

The only change to the API of module `SimPy.Simulation` is the addition of class `Simulation`:

```
1  Module SimPy.Simulation:
2     ################ Unchanged ###############
3     ## yield-verb constants --------------------
4     get
5     hold
6     passivate
7     put
8     queueevent
9     release
10    request
11    waitevent
12    waituntil
13    ## version constant ------------------------
14    version
15    ## classes ---------------------------------
16    FatalSimerror
17    Simerror
18    ################ Added ###############
19    Simulation
```

Thus, after the import:

```
from SimPy.Simulation import *
```

class `Simulation` is available to a program.

Actually,:

```
from SimPy.Simulation import Simulation
```

is sufficient and even clearer.

class Simulation

The simulation capabilities of a model are provided by instantiating class Simulation like this:

```
from SimPy.Simulation import *

aSimulation = Simulation()
## model code follows
```

Better OO programming style is actually to define a model class which inherits from Simulation:

```
import SimPy.Simulation as Simulation

class MyModel(Simulation.Simulation):
    def run(self):
        self.initialize()
        ## model code follows

myMo = MyModel()
myMo.run()
```

The self.initialize() is not really necessary, as the Simulation instance is initialized at generation time. If method run for a model (here myMo) is executed more than once, e.g. for running a simulation repeatedly, self.initialize() resets the model to an empty event list and simulation time 0.

Methods of class Simulation

class Simulation has these methods:

```
class Simulation:
  ## Methods -----------------------------------
  __init__(self)
  initialize(self)
  now(self)
  stopSimulation(self)
  allEventNotices(self)
  allEventTimes(self)
  activate(self, obj, process, at='undefined', delay='undefined', prior=False)
  reactivate(self, obj, at='undefined', delay='undefined', prior=False)
  startCollection(self, when=0.0, monitors=None, tallies=None)
  simulate(self, until=0)
```

The semantics and parameters (except for self) of the methods are identical to those of the non-OO SimPy.Simulation functions of the same name. For example, to get the current simulation time of a Simulation object so, the call is:

```
tcurrent = so.now()
```

Module SimPy.SimulationTrace

The only change to the API of module SimPy.SimulationTrace is the addition of class SimulationTrace:

```
Module SimPy.SimulationTrace:
  ############### Unchanged ###############
  ## yield-verb constants --------------------
```

```
  get
  hold
  passivate
  put
  queueevent
  release
  request
  waitevent
  waituntil
  ## version constant -------------------------
  version
  ## classes ---------------------------------
  FatalSimerror
  Simerror
  Trace
  ################ Added ###############
  SimulationTrace
```

class SimulationTrace

The simulation capabilities of a model with tracing are provided by instantiating class `SimulationTrace` like this:

```
from SimPy.SimulationTrace import SimulationTrace

aSimulation = SimulationTrace()
## model code follows
```

Again, better OO programming style is actually to define a model class which inherits from Simulation:

```
from SimPy.SimulationTrace import SimulationTrace

class MyModel(SimulationTrace):
    def run(self):
        self.initialize()
        # model code follows

myMo = MyModel()
myMo.run()
```

class `SimulationTrace` is a subclass of `Simulation` and thus provides the same methods, albeit with tracing added.

The semantics and parameters of the methods are identical to those of the non-OO `SimPy.SimulationTrace` functions of the same name.

Methods and attributes of class SimulationTrace

```
class SimulationTrace:
   ## Methods ----------------------------------
   __init__(self)
   initialize(self)
   now(self)
   stopSimulation(self)
   allEventNotices(self)
```

```
    allEventTimes(self)
    activate(self, obj, process, at='undefined', delay='undefined', prior=False)
    reactivate(self, obj, at='undefined', delay='undefined', prior=False)
    startCollection(self, when=0.0, monitors=None, tallies=None)
    simulate(self, until=0)
    ## trace attribute --------------------------
    trace
```

Attribute `trace`

An initialization of class `SimulationTrace` generates an instance of class `Trace`. This becomes an attribute `trace` of the `SimulationTrace` instance.

`Trace` methods

The semantics and parameters of the `Trace` methods are identical to those of the non-OO `SimPy.SimulationTrace` `trace` instance of the same name.

- trace.start(self)

Example:

```
s.trace.start()
```

- trace.stop(self)
- trace.treset(self)
- trace.tchange(self, **kmvar)
- trace.ttext(self,par)

Example calls (snippet):

```
from SimPy.SimulationTrace import SimulationTrace
s = SimulationTrace()
s.initialize()
s.trace.ttext("Here we go")
```

Again, note that you have to qualify the `trace` instance (see e.g. the last line of the snippet) with the `SimulationTrace` instance, here `s`.

Module `SimPy.SimulationRT`

class `SimulationRT`

The simulation capabilities plus real time synchronization are provided by instantiating class `SimulationRT`.

Methods of class `SimulationRT`

The `SimulationRT` subclass adds two methods to those inherited from `Simulation`.

The semantics and parameters of the methods are identical to those of the non-OO `SimPy.SimulationRT` functions of the same name.

- rtnow
- rtset

Example calls (snippet):

```
from SimPy.SimulationRT import Process, hold
class Car(Process):
   def __init__(self):
      Process.__init__(self, sim=self.sim)
   def run(self):
      print(self.sim.rtnow())
      yield hold, self, 10
```

class SimulationStep

The simulation capabilities plus event stepping are provided by instantiating class SimulationStep.

Methods of class SimulationStep

The SimulationStep subclass adds three methods to those inherited from Simulation.

The semantics and parameters of the methods are identical to those of the non-OO SimPy.SimulationStep functions of the same name.

- startStepping
- stopStepping
- simulateStep

Example call (snippet):

```
from SimPy.SimulationStep import *
s = SimulationStep()
s.initialize()
s.simulateStep(until=100, callback=myCallBack)
```

Classes with a SimulationXX attribute

All SimPy entity (*Process*, *Resource*, *Store*, *Level*, *SimEvent*) and monitoring (*Monitor*, *Tally*) classes have time-related functions. In the OO-API of SimPy, they therefore have a .sim attribute which is a reference to the *SimulationXX* instance to which they belong. This association is made by providing that reference as a parameter to the constructor of the class.

Important: All class instances instances must refer to the same SimulationXX instance, i.e., their .sim attributes must have the same value. That value must be the reference to the SimulationXX instance. Any deviation from this will lead to strange mis-functioning of a SimPy script.

The constructor calls (signatures) for the classes in question thus change as follows:

class Process

```
Process.__init__(self, name = 'a_process', sim = None)
```

Example 1 (snippet):

```
class Car(Process):
    def drive(self):
        yield hold, self, 10
        print("Arrived at", self.sim.now())

aSim = Simulation()
aSim.initialize()
c=Car(name="Mine", sim=aSim)
```

Example 2, with an __init__ method (snippet):

```
class Car(Process):
   def __init__(self, name):
        Process.__init__(self, name=name, sim=self.sim)

aSim = Simulation()
aSim.initialize()
c=Car(name="Mine", whichSim=aSim)
```

class Resource

```
Resource.__init__(self, capacity=1, name='a_resource',
                  unitName='units',
                  qType=FIFO, preemptable=0, monitored=False,
                  monitorType=Monitor, sim=None)
```

Example (snippet):

```
aSim = Simulation()
aSim.initialize()
res = Resource(name="Server", sim=aSim)
```

classes Store and Level

Store:

```
Store.__init__(self, name=None, capacity='unbounded', unitName='units',
               putQType=FIFO, getQType=FIFO,
               monitored=False, monitorType=Monitor, initialBuffered=None,
               sim=None)
```

Level:

```
Level.__init__(self, name=None, capacity='unbounded', unitName='units',
               putQType=FIFO, getQType=FIFO,
               monitored=False, monitorType=Monitor, initialBuffered=None,
               sim=None)
```

Example (snippet):

```
aSim = Simulation()
aSim.initialize()
buffer = Store(name="Parts", sim=aSim)
```

class SimEvent

```
SimEvent.__init__(self, name='a_SimEvent', sim=None)
```

Example (snippet):

```
aSim = Simulation()
aSim.initialize()
evt = SimEvent("Boing!", sim=aSim)
```

classes Monitor and Tally

Monitor:

```
Monitor.__init__(self, name='a_Monitor', ylab='y', tlab='t', sim=None)
```

Tally:

```
Tally.__init__(self, name='a_Tally', ylab='y', tlab='t', sim=None)
```

Example (snippet):

```
aSim = Simulation()
aSim.initialize()
myMoni = Monitor(name="Counting cars", sim=aSim)
```

2.3 SimPy Classic Simulation with Tracing

Authors

- Klaus Muller <Muller@users.sourceforge.net>

Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7 and later

Date December 2011

Updated January 2018

Contents

2.3.1 Introduction

The tracing utility has been developed to give users insight into the dynamics of the execution of SimPy simulation programs. It can help developers with testing and users with explaining SimPy models to themselves and others (e.g. for documentation or teaching purposes).

2.3.2 Tracing *SimPy* programs

Tracing any *SimPy* program is as simple as replacing:

```
from SimPy.Simulation import *
```

with:

```
from SimPy.SimulationTrace import *
```

This will give a complete trace of all the scheduling statements executed during the program's execution.

An even nicer way is to replace this import by:

```
if __debug__:
    from SimPy.SimulationTrace import *
else:
    from SimPy.Simulation import *
```

This gives a trace during the development and debugging. If one then executes the program with *python -O myprog.py*, tracing is switched off, and no run-time overhead is incurred. (*__debug__* is a global Python constant which is set to False by commandline options -O and -OO.)

For the same reason, any user call to *trace* methods should be written as:

```
if __debug__:
    trace.ttext("This will only show during debugging")
```

Here is an example (bank02.py from the Bank Tutorial):

```
import SimPy.SimulationTrace as Simulation  # <== changed for tracing
# import SimPy.Simulation as Simulation


""" Simulate a single customer """


class Customer(Simulation.Process):
    """ Customer arrives, looks around and leaves """
```

```
    def __init__(self, name):
        Simulation.Process.__init__(self)
        self.name = name

    def visit(self, timeInBank=0):
        print("%7.4f %s: Here I am" % (Simulation.now(), self.name))
        yield Simulation.hold, self, timeInBank
        print("%7.4f %s: I must leave" % (Simulation.now(), self.name))


def model():
    Simulation.initialize()
    c1 = Customer(name="Klaus")
    Simulation.activate(c1, c1.visit(timeInBank=10.0), delay=5.0)
    c2 = Customer(name="Tony")
    Simulation.activate(c2, c2.visit(timeInBank=8.0), delay=2.0)
    c3 = Customer(name="Evelyn")
    Simulation.activate(c3, c3.visit(timeInBank=20.0), delay=12.0)
    Simulation.simulate(until=400.0)


model()
```

This program produces the following output:

```
0 activate <Klaus> at time: 5.0 prior: False
0 activate <Tony> at time: 2.0 prior: False
0 activate <Evelyn> at time: 12.0 prior: False
 2.0000 Tony: Here I am
2.0 hold <Tony> delay: 8.0
 5.0000 Klaus: Here I am
5.0 hold <Klaus> delay: 10.0
10.0000 Tony: I must leave
10.0 <Tony> terminated
12.0000 Evelyn: Here I am
12.0 hold <Evelyn> delay: 20.0
15.0000 Klaus: I must leave
15.0 <Klaus> terminated
32.0000 Evelyn: I must leave
32.0 <Evelyn> terminated
```

Another example:

```
""" bank09.py: Simulate customers arriving
    at random, using a Source requesting service
    from several clerks but a single queue
    with a random servicetime
"""
from __future__ import generators
from random import Random
import SimPy.SimulationTrace as Simulation


class Source(Simulation.Process):
    """ Source generates customers randomly"""

    def __init__(self, seed=333):
```

```
        Simulation.Process.__init__(self)
        self.SEED = seed

    def generate(self, number, interval):
        rv = Random(self.SEED)
        for i in range(number):
            c = Customer(name="Customer%02d" % (i,))
            Simulation.activate(c, c.visit(timeInBank=12.0))
            t = rv.expovariate(1.0 / interval)
            yield Simulation.hold, self, t


class Customer(Simulation.Process):
    """ Customer arrives, is served and leaves """

    def __init__(self, name):
        Simulation.Process.__init__(self)
        self.name = name

    def visit(self, timeInBank=0):
        arrive = Simulation.now()
        print("%7.4f %s: Here I am " % (Simulation.now(), self.name))
        yield Simulation.request, self, counter
        wait = Simulation.now() - arrive
        print("%7.4f %s: Waited %6.3f" % (Simulation.now(),
                                          self.name, wait))
        tib = counterRV.expovariate(1.0 / timeInBank)
        yield Simulation.hold, self, tib
        yield Simulation.release, self, counter
        print("%7.4f %s: Finished" % (Simulation.now(), self.name))


def model(counterseed=3939393):
    global counter, counterRV
    counter = Simulation.Resource(name="Clerk", capacity=2)  # Lcapacity
    counterRV = Random(counterseed)
    Simulation.initialize()
    sourceseed = 1133
    source = Source(seed=sourceseed)
    Simulation.activate(source, source.generate(5, 10.0), 0.0)
    Simulation.simulate(until=400.0)


model()
```

This produces:

```
0 activate <a_process> at time: 0 prior: 0
0 activate <Customer00> at time: 0 prior: 0
0 hold <a_process> delay: 8.73140489458
 0.0000 Customer00: Here I am
0 request <Customer00> <Clerk>  priority: default
. . .waitQ: []
. . .activeQ: ['Customer00']
 0.0000 Customer00: Waited  0.000
0 hold <Customer00> delay: 8.90355092634
8.73140489458 activate <Customer01> at time: 8.73140489458 prior: 0
8.73140489458 hold <a_process> delay: 8.76709801376
```

```
 8.7314 Customer01: Here I am
8.73140489458 request <Customer01> <Clerk>  priority: default
. . .waitQ: []
. . .activeQ: ['Customer00', 'Customer01']
 8.7314 Customer01: Waited  0.000
8.73140489458 hold <Customer01> delay: 21.6676883425
8.90355092634 release <Customer00> <Clerk>
. . .waitQ: []
. . .activeQ: ['Customer01']
 8.9036 Customer00: Finished
8.90355092634 <Customer00> terminated
17.4985029083 activate <Customer02> at time: 17.4985029083 prior: 0

. . . . .
```

And here is an example showing the trace output for compound yield statements:

```
import SimPy.SimulationTrace as Simulation


class Client(Simulation.Process):
    def __init__(self, name):
        Simulation.Process.__init__(self, name)

    def getServed(self, tank):
        yield (Simulation.get, self, tank, 10), (Simulation.hold, self, 1.5)
        if self.acquired(tank):
            print("%s got 10 %s" % (self.name, tank.unitName))
        else:
            print("%s reneged" % self.name)


class Filler(Simulation.Process):
    def __init__(self, name):
        Simulation.Process.__init__(self, name)

    def fill(self, tank):
        for i in range(3):
            yield Simulation.hold, self, 1
            yield Simulation.put, self, tank, 10


Simulation.initialize()
tank = Simulation.Level(name="Tank", unitName="gallons")
for i in range(2):
    c = Client("Client %s" % i)
    Simulation.activate(c, c.getServed(tank))
f = Filler("Tanker")
Simulation.activate(f, f.fill(tank))
Simulation.simulate(until=10)
```

It produces this output:

```
0 activate <Client 0> at time: 0 prior: False
0 activate <Client 1> at time: 0 prior: False
0 activate <Tanker> at time: 0 prior: False
0 activate <RENEGE - hold for Client 0> at time: 0 prior: False
0 get <Client 0>to get: 10 gallons from <Tank> priority: default
```

```
. . .getQ: ['Client 0']
. . .putQ: []
. . .in buffer: 0
|| RENEGE COMMAND:
||          hold <Client 0> delay: 1.5
0 activate <RENEGE - hold for Client 1> at time: 0 prior: False
0 get <Client 1>to get: 10 gallons from <Tank> priority: default
. . .getQ: ['Client 0', 'Client 1']
. . .putQ: []
. . .in buffer: 0
|| RENEGE COMMAND:
||          hold <Client 1> delay: 1.5
0 hold <Tanker> delay: 1
0 hold <RENEGE - hold for Client 0> delay: 1.5
0 hold <RENEGE - hold for Client 1> delay: 1.5
1 put <Tanker> to put: 10 gallons into <Tank> priority: default
. . .getQ: ['Client 1']
. . .putQ: []
. . .in buffer: 0
1 hold <Tanker> delay: 1
Client 0 got 10 gallons
1 <Client 0> terminated
1.5 reactivate <Client 1> time: 1.5 prior: False
1.5 <RENEGE - hold for Client 1> terminated
Client 1 reneged
1.5 <Client 1> terminated
2 put <Tanker> to put: 10 gallons into <Tank> priority: default
. . .getQ: []
. . .putQ: []
. . .in buffer: 10
2 hold <Tanker> delay: 1
3 put <Tanker> to put: 10 gallons into <Tank> priority: default
. . .getQ: []
. . .putQ: []
. . .in buffer: 20
3 <Tanker> terminated
```

In this example, the Client entities are requesting 10 gallons from the *tank* (a Level object). If they can't get them within 1.5 time units, they renege (give up waiting). The renege command parts of the compound statements (*hold,self,1.5*)are shown in the trace output with a prefix of || to indicate that they are being executed in parallel with the primary command part (*get,self,tank,10*). They are being executed by behind-the-scenes processes (e.g. *RENEGE-hold for Client 0*).

The trace contains all calls of scheduling statements (**yield ...**, **activate()**, **reactivate()**, **cancel()** and also the termination of processes (at completion of all their scheduling statements). For **yield request** and **yield release** calls, it provides also the queue status (waiting customers in *waitQ* and customers being served in *activeQ*.

2.3.3 trace.tchange(): Changing the tracing

trace is an instance of the **Trace** class defined in *SimulationTrace.py*. This gets automatically initialized upon importing *SimulationTrace*..

The tracing can be changed at runtime by calling **trace.tchange()** with one or more of the following named parameters:

start:

changes the tracing start time. Default is 0. Example: **trace.tchange(start=222.2)** to start tracing at simulation time 222.2.

end :

changes the tracing end time. Default is a very large number (hopefully past any simulation endtime you will ever use). Example: **trace.tchange(end=33)** to stop tracing at time 33.

toTrace:

changes the commands to be traced. Default is *["hold","activate","cancel","reactivate","passivate","request", "release","interrupt","waitevent","queueevent", "signal","waituntil","put","get","terminated"]*. Value must be a list containing one or more of those values in the default. Note: "terminated" causes tracing of all process terminations. Example: **trace.tchange(toTrace=["hold","activate"])** traces only the *yield hold* and *activate()* statements.

outfile:

redirects the trace out put to a file (default is *sys.stdout*). Value must be a file object open for writing. Example: **trace.tchange(outfile=open(r"c:\python25\bank02trace.txt","w"))**

All these parameters can be combined. Example: **trace.tchange(start=45.0,toTrace=["terminated"])** will trace all process terminations from time 45.0 till the end of the simulation.

The changes become effective at the time **trace.tchange()** is called. This implies for example that, if the call **trace.tchange(start=50)** is made at time 100, it has no effect before *now()==100*.

2.3.4 treset(): Resetting the trace to default values

The trace parameters can be reset to their default values by calling **trace.treset()**.

2.3.5 trace.tstart(), trace.tstop(): Enabling/disabling the trace

Calling **trace.tstart()** enables the tracing, and **trace.tstop()** disables it. Neither call changes any tracing parameters.

2.3.6 trace.ttext(): Annotating the trace

The event-by-event trace output is already very useful in showing the sequence in which SimPy's quasi-parallel processes are executed.

For documentation, publishing or teaching purposes, it is even more useful if the trace output can be intermingled with output which not only shows the command executed, but also contextual information such as the values of state variables. If one outputs the reason *why* a specific scheduling command is executed, the trace can give a natural language description of the simulation scenario.

For such in-line annotation, the **trace.ttext(<string>)** method is available. It provides a string which is output together with the trace of the next scheduling statement. This string is valid *only* for the scheduling statement following it.

Example:

```
import SimPy.SimulationTrace as Simulation


class Bus(Simulation.Process):
    def __init__(self, name):
        Simulation.Process.__init__(self, name)

    def operate(self, repairduration=0):
        tripleft = 1000
```

```
        while tripleft > 0:
            Simulation.trace.ttext("Try to go for %s" % tripleft)
            yield Simulation.hold, self, tripleft
            if self.interrupted():
                tripleft = self.interruptLeft
                self.interruptReset()
                Simulation.trace.ttext("Start repair taking %s time units" %
                                       repairduration)
                yield Simulation.hold, self, repairduration
            else:
                break  # no breakdown,  ergo bus arrived
        Simulation.trace.ttext("<%s> has arrived" % self.name)


class Breakdown(Simulation.Process):
    def __init__(self, myBus):
        Simulation.Process.__init__(self, name="Breakdown " + myBus.name)
        self.bus = myBus

    def breakBus(self, interval):

        while True:
            Simulation.trace.ttext("Breakdown process waiting for %s" %
                                   interval)
            yield Simulation.hold, self, interval
            if self.bus.terminated():
                break
            Simulation.trace.ttext("Breakdown of %s" % self.bus.name)
            self.interrupt(self.bus)


print("\n\n+++test_interrupt")
Simulation.initialize()
b = Bus("Bus 1")
Simulation.trace.ttext("Start %s" % b.name)
Simulation.activate(b, b.operate(repairduration=20))
br = Breakdown(b)
Simulation.trace.ttext("Start the Breakdown process for %s" % b.name)
Simulation.activate(br, br.breakBus(200))
Simulation.trace.start = 100
print(Simulation.simulate(until=4000))
```

This produces:

```
+++test_interrupt
0 activate <Bus 1> at time: 0 prior: False
---- Start Bus 1
0 activate <Breakdown Bus 1> at time: 0 prior: False
---- Start the Breakdown process for Bus 1
200 reactivate <Bus 1> time: 200 prior: False
200 interrupt by: <Breakdown Bus 1> of: <Bus 1>
---- Breakdown of Bus 1
200 hold <Breakdown Bus 1> delay: 200
---- Breakdown process waiting for 200
200 hold <Bus 1> delay: 20
---- Start repair taking 20 time units
220 hold <Bus 1> delay: 800
---- Try to go for 800
```

```
400 reactivate <Bus 1> time: 400 prior: False
400 interrupt by: <Breakdown Bus 1> of: <Bus 1>
---- Breakdown of Bus 1
400 hold <Breakdown Bus 1> delay: 200

 . . . . .
```

The line starting with "—-" is the comment related to the command traced in the preceding output line.

2.3.7 Nice output of class instances

After the import of *SimPy.SimulationTrace*, all instances of classes *Process* and *Resource* (and all their subclasses) have a nice string representation like so:

```
   >>> class Bus(SimulationProcess):
...         def __init__(self, id):
...                 Simulation.Process.__init__(self, name=id)
...                 self.typ = "Bus"
...
>>> b = Bus("Line 15")
>>> b
<Instance of Bus, id 21860960:
     .name=Line 15
     .typ=Bus
>
>>>
```

This can be handy in statements like **trace.ttext("Status of %s"%b)**.

2.4 Simulation with Real Time Synchronization

Authors

- Klaus Muller <Muller@users.sourceforge.net>
- Tony Vignaux <Vignaux@users.sourceforge.net>

Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7+

Date December 2011

Updated January 2018

Contents

This manual describes **SimulationRT**, a SimPy module which supports synchronizing the execution of simulation models with real (wallclock) time.

2.4.1 Acknowledgement

SimulationRT is based on an idea by Geoff Jarrad of CSIRO (Australia). He contributed a lot to its development and testing on Windows and Unix.

The code for the adjustment of the execution speed during the simulation run was contributed by Robert C. Ramsdell.

2.4.2 Synchronizing with wall clock time

SimulationRT allows synchronizing simulation time and real (wallclock) time. This capability can be used to implement e.g. interactive game applications or to demonstrate a model's execution in real time.

It is identical to Simulation, except for the *simulate* function which takes an additional parameter controlling real-time execution speed.

Here is an example:

```
""" RealTimeFireworks.py """
import SimPy.SimulationRT as SimulationRT
from random import seed, uniform
import time


# Model components ------------------------------------------------------------
class Launcher(SimulationRT.Process):
    def launch(self):
        while True:
            print("Launch at %2.4f; wallclock: %2.4f" %
                  (SimulationRT.now(),
                   time.clock() - startTime))
            yield SimulationRT.hold, self, uniform(1, maxFlightTime)
            print("Boom!!! Aaaah!! at %2.4f; wallclock: %2.4f" %
                  (SimulationRT.now(), time.clock() - startTime))


def model():
    SimulationRT.initialize()
    for i in range(nrLaunchers):
        lau = Launcher()
        SimulationRT.activate(lau, lau.launch())
    SimulationRT.simulate(
        real_time=True, rel_speed=1, until=20)  # unit sim time = 1 sec clock


# Experiment data  ------------------------------------------------------------
```

```
nrLaunchers = 2
maxFlightTime = 5.0
startTime = time.clock()
seed(1234567)
# Experiment --------------------------------------------------------------
model()
```

rel_speed is the ratio **simulated time/wallclock time**. *rels_speed=1* sets the synchronization so that 1 simulation time unit is executed in approximately 1 second of wallclock time. Run under Python 2.6 on a Windows Vista-box (2.3 GHz), this output resulted over about 17.5 seconds of wallclock time:

```
Launch at 0.00; wallclock: 0.00
Launch at 0.00; wallclock: 0.00
Boom!!! Aaaah!! at 1.94; wallclock: 0.00
Launch at 1.94; wallclock: 0.00
Boom!!! Aaaah!! at 4.85; wallclock: 0.00
Launch at 4.85; wallclock: 0.00
Boom!!! Aaaah!! at 5.27; wallclock: 0.00
Launch at 5.27; wallclock: 0.00
Boom!!! Aaaah!! at 6.76; wallclock: 0.00
Launch at 6.76; wallclock: 0.00
Boom!!! Aaaah!! at 10.14; wallclock: 0.00
Launch at 10.14; wallclock: 0.00
Boom!!! Aaaah!! at 10.21; wallclock: 0.00
Launch at 10.21; wallclock: 0.00
Boom!!! Aaaah!! at 11.43; wallclock: 0.00
Launch at 11.43; wallclock: 0.00
Boom!!! Aaaah!! at 13.34; wallclock: 0.00
Launch at 13.34; wallclock: 0.00
Boom!!! Aaaah!! at 14.85; wallclock: 0.00
Launch at 14.85; wallclock: 0.00
Boom!!! Aaaah!! at 17.48; wallclock: 0.00
Launch at 17.48; wallclock: 0.00
Boom!!! Aaaah!! at 19.15; wallclock: 0.00
Launch at 19.15; wallclock: 0.00
Boom!!! Aaaah!! at 19.18; wallclock: 0.00
Launch at 19.18; wallclock: 0.00
```

Clearly, the wallclock time does not deviate significantly from the simulation time.

2.4.3 Changing the execution speed during a simulation run

By calling method *rtset* with a parameter, the ratio simulated time to wallclock time can be changed during a run.

Here is an example:

```
"""variableTimeRatio.py
Shows the SimulationRT capability to change the ratio simulation
time to wallclock time during the run of a simulation.
"""
import SimPy.SimulationRT as SimulationRT


class Changer(SimulationRT.Process):
    def change(self, when, rat):
        global ratio
        yield SimulationRT.hold, self, when
```

```
        SimulationRT.rtset(rat)
        ratio = rat


class Series(SimulationRT.Process):
    def tick(self, nrTicks):
        oldratio = ratio
        for i in range(nrTicks):
            tLastSim = SimulationRT.now()
            tLastWallclock = SimulationRT.wallclock()
            yield SimulationRT.hold, self, 1
            diffSim = SimulationRT.now() - tLastSim
            diffWall = SimulationRT.wallclock() - tLastWallclock
            print("now(): %s, sim. time elapsed: %s, wall clock elapsed: "
                  "%6.3f, sim/wall time ratio: %6.3f" %
                  (SimulationRT.now(), diffSim, diffWall, diffSim / diffWall))
            if not ratio == oldratio:
                print("At simulation time %s: ratio simulation/wallclock "
                      "time now changed to %s" % (SimulationRT.now(), ratio))
                oldratio = ratio


SimulationRT.initialize()
ticks = 15
s = Series()
SimulationRT.activate(s, s.tick(nrTicks=ticks))
c = Changer()
SimulationRT.activate(c, c.change(5, 5))
c = Changer()
SimulationRT.activate(c, c.change(10, 10))
ratio = 1
print("At simulation time %s: set ratio simulation/wallclock time to %s" %
      (SimulationRT.now(), ratio))
SimulationRT.simulate(until=100, real_time=True, rel_speed=ratio)
```

The program changes the time ratio twice, at simulation times 5 and 10.

When run on a Windows Vista computer under Python 2.7, this results in this output:

```
At simulation time 0: set ratio simulation/wallclock time to 1
now(): 1, sim. time elapsed: 1, wall clock elapsed:  0.998, sim/wall time ratio:  1.
↪002
now(): 2, sim. time elapsed: 1, wall clock elapsed:  0.999, sim/wall time ratio:  1.
↪001
now(): 3, sim. time elapsed: 1, wall clock elapsed:  0.999, sim/wall time ratio:  1.
↪001
now(): 4, sim. time elapsed: 1, wall clock elapsed:  0.999, sim/wall time ratio:  1.
↪001
now(): 5, sim. time elapsed: 1, wall clock elapsed:  0.999, sim/wall time ratio:  1.
↪001
At simulation time 5: ratio simulation/wallclock time now changed to 5
now(): 6, sim. time elapsed: 1, wall clock elapsed:  0.199, sim/wall time ratio:  5.
↪027
now(): 7, sim. time elapsed: 1, wall clock elapsed:  0.199, sim/wall time ratio:  5.
↪025
now(): 8, sim. time elapsed: 1, wall clock elapsed:  0.199, sim/wall time ratio:  5.
↪026
now(): 9, sim. time elapsed: 1, wall clock elapsed:  0.199, sim/wall time ratio:  5.
↪026
```

```
now(): 10, sim. time elapsed: 1, wall clock elapsed:  0.199, sim/wall time ratio:  5.
↪024
At simulation time 10: ratio simulation/wallclock time now changed to 10
now(): 11, sim. time elapsed: 1, wall clock elapsed:  0.099, sim/wall time ratio: 10.
↪108
now(): 12, sim. time elapsed: 1, wall clock elapsed:  0.099, sim/wall time ratio: 10.
↪105
now(): 13, sim. time elapsed: 1, wall clock elapsed:  0.099, sim/wall time ratio: 10.
↪102
now(): 14, sim. time elapsed: 1, wall clock elapsed:  0.099, sim/wall time ratio: 10.
↪104
now(): 15, sim. time elapsed: 1, wall clock elapsed:  0.099, sim/wall time ratio: 10.
↪104
```

2.4.4 Limitations

This module works much better under Windows than under Unix or Linux, i.e., it gives much closer synchronization. Unfortunately, the handling of time in Python is not platform-independent at all. Here is a quote from the documentation of the *time* module:

```
"clock()
On Unix, return the current processor time as a floating point number expressed in␣
↪seconds.
The precision, and in fact the very definition of the meaning of ``processor time'' ,␣
↪depends
on that of the C function of the same name, but in any case, this is the function to␣
↪use for
benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to␣
↪this
function, as a floating point number, based on the Win32 function␣
↪QueryPerformanceCounter().
The resolution is typically better than one microsecond.
"
```

Also it is deprecated in 3.3 and up.

2.4.5 The SimulationRT API

Structure

Basically, SimulationStep has the same API as Simulation, but with:

- a change in the definition of simulate, and
- an additional method to change execution speed during a simulation run.

simulate

Executes the simulation model.

Call:

simulate(<optional parameters>)

Mandatory parameters: None.

Optional parameters:

- **until=0** : the maximum simulation (end) time (positive floating point number; default: 0)
- **real_time=False** : flag to switch real time synchronization on or off (boolean; default: False, meaning no synchronization)
- **rel_speed=1** : ratio simulation time over wallclock time; example: *rel_speed=200* executes 200 units of simulation time in about one second (positive floating point number; default: 1, i.e. 1 sec of simulation time is executed in about 1 sec of wallclock time)

Return value: Simulation status at exit.

rtset

Changes the ratio simulation time over wall clock time.

Call:

rtset(<new ratio>)

Mandatory parameters: None

Optional parameters:

- **rel_speed=1** : ratio simulation time over wallclock time; example: *rel_speed=200* executes 200 units of simulation time in about one second (positive floating point number; default: 1, i.e. 1 sec of simulation time is executed in about 1 sec of wallclock time)

Return value: None

2.5 SimPy Classic Simulation with Event Stepping

Authors

- Klaus Muller <Muller@users.sourceforge.net>
- Tony Vignaux <Vignaux@users.sourceforge.net>

Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python Version 2.7+

Date December 2011

Updated January 2018

Contents

This manual describes **SimulationStep**, a SimPy module which supports stepping through a simulation model event by event.

2.5.1 Introduction

SimulationStep can assist with debugging models, interacting with them on an event-by-event basis, getting event-by-event output from a model (e.g. for plotting purposes), etc.

SimulationStep is a derivative of the Simulation module. Over and above the capabilities provided by Simulation, SimulationStep supports stepping through a simulation model event by event. It caters for:

- running a simulation model, with calling a user-defined procedure after every event,
- running a simulation model one event at a time by repeated calls,
- starting and stopping the event stepping mode under program control.

2.5.2 SimulationStep overview

Here is a simple program which shows basic event stepping capabilities:

```
# simstep_stepping1.py
import SimPy.SimulationStep as SimulationStep        # (1)


def callbackTimeTrace():                             # (2)
    """Prints event times
    """
    print("at time=%s" % SimulationStep.now())


class Man(SimulationStep.Process):
    def walk(self):
        print("got up")
        yield SimulationStep.hold, self, 1
        print("got to door")
        yield SimulationStep.hold, self, 10
        print("got to mail box")
        yield SimulationStep.hold, self, 10
        print("got home again")


# trace event times
SimulationStep.initialize()
otto = Man()
SimulationStep.activate(otto, otto.walk())
SimulationStep.startStepping()                       # (3)
SimulationStep.simulate(callback=callbackTimeTrace, until=100)   # (4)
```

A trivial simulation model, but with event stepping:

1. import the stepping version of Simulation
2. define a procedure which gets called after every event
3. switch into event stepping mode
4. run the model with event callback to the procedure defined at (2); `simulate` in SimulationStep has an extra named parameter, `callback`.

Running it produces this output:

```
got up
at time=0
got to door
at time=1
got to mail box
at time=11
got home again
at time=21
```

The callback outputs the simulation time after every event.

Here is another example, the same model, but now with the user getting control back after every event:

```
# simstep_stepping2.py
import SimPy.SimulationStep as SimulationStep


def callbackUserControl():
    """Allows user to control stepping
    """
    # In python 2.7 you need to make this raw_input
    a = input("[Time=%s] Select one: End run (e), Continue stepping (s), "
              "Run to end (r)= " % SimulationStep.now())
    if a == "e":
        SimulationStep.stopSimulation()
    elif a == "s":
        return
    else:
        SimulationStep.stopStepping()


class Man(SimulationStep.Process):
    def walk(self):
        print("got up")
        yield SimulationStep.hold, self, 1
        print("got to door")
        yield SimulationStep.hold, self, 10
        print("got to mail box")
        yield SimulationStep.hold, self, 10
        print("got home again")


# allow user control
SimulationStep.initialize()
otto = Man()
SimulationStep.activate(otto, otto.walk())
SimulationStep.startStepping()
SimulationStep.simulate(callback=callbackUserControl, until=100)
```

Its interactive output looks like this:

```
got up
[Time=0] Select one: End run (e), Continue stepping (s), Run to end (r)= s
got to door
[Time=1] Select one: End run (e), Continue stepping (s), Run to end (r)= s
got to mail box
[Time=11] Select one: End run (e), Continue stepping (s), Run to end (r)= s
```

```
got home again
[Time=21] Select one: End run (e), Continue stepping (s), Run to end (r)= s
[Time=21] Select one: End run (e), Continue stepping (s), Run to end (r)= s
```

or this (the user stopped stepping mode at time=1):

```
got up
[Time=0] Select one: End run (e), Continue stepping (s), Run to end (r)= s
got to door
[Time=1] Select one: End run (e), Continue stepping (s), Run to end (r)= r
got to mail box
got home again
```

If one wants to run a tested/debugged model full speed, i.e. without stepping, one can write a program as follows:

```
# simstep_stepping2fast.py

if __debug__:
    import SimPy.SimulationStep as Simulation
else:
    import SimPy.Simulation as Simulation


def callbackUserControl():
    """Allows user to control stepping
    """
    if __debug__:
        # In python 2.7 you need to make this raw_input
        a = input("[Time=%s] Select one: End run (e), Continue stepping (s),"
                  "Run to end (r)= " % Simulation.now())
        if a == "e":
            Simulation.stopSimulation()
        elif a == "s":
            return
        else:
            Simulation.stopStepping()


class Man(Simulation.Process):
    def walk(self):
        print("got up")
        yield Simulation.hold, self, 1
        print("got to door")
        yield Simulation.hold, self, 10
        print("got to mail box")
        yield Simulation.hold, self, 10
        print("got home again")


# allow user control if debugging
Simulation.initialize()
otto = Man()
Simulation.activate(otto, otto.walk())
if __debug__:
    Simulation.startStepping()
    Simulation.simulate(callback=callbackUserControl, until=100)
else:
    Simulation.simulate(until=100)
```

If one runs this with the Python command line option '-O', any statement starting with `if __debug__:` is ignored/skipped by the Python interpreter.

2.5.3 The SimulationStep API

Structure

Basically, SimulationStep has the same API as Simulation, but with the following additions and changes:

```
def startStepping()          **new**
def stopStepping()           **new**
def simulate()               **changed**
def simulateStep()           **new**
```

startStepping

Starts the event-stepping.

Call:

startStepping()

Mandatory parameters: None.

Optional parameters: None

Return value: None.

stopStepping

Stops event-stepping.

Call: stopStepping()

Mandatory parameters: None

Optional parameters: None

Return value: None

simulate

Runs a simulation with callback to a user-defined function after each event, if stepping is turned on. By default, stepping is switched off.

Call: simulate(callback=<proc>,until=<endtime>)

Mandatory parameters: None

Optional parameters:

- **until = 0**: the simulation time until which the simulation is to run (positive floating point or integer number)
- **callback = lambda:None**: the function to be called after every event (function reference)

Return value: The simulation status at exit (string)

simulateStep

Runs a simulation for one event, with (optional) callback to a user-defined function after the event, if stepping is turned on. By default, stepping is switched off. Thus, to execute the model to completion, *simulateStep* must be called repeatedly.

Note: it is not yet clear to the developers whether this part of the API offers any advantages or capabilities over and above the *simulate* function. The survival of this function in future versions depends on the feedback from the user community.

Call: simulateStep(callback=<proc>,until=<endtime>)

Mandatory parameters: None

Optional parameters:

- **until = 0**: the simulation time until which the simulation is to run (positive floating point or integer number)
- **callback = lambda:None**: the function to be called after every event (function reference)

Return value: The tuple **(simulation status at exit (string),<resumability flag>)**. <resumability flag> can have one of two string values: **"resumable"** if there are more events to be executed, and **"notResumable"** if all events have been exhausted or an error has occurred. *simulateStep* should normally only be called if "resumable" is returned.

2.6 Cheatsheets

If you want to have a list of all SimPy commands, their syntax and parameters for your desktop, print out a copy of the **SimPy Cheatsheet**, or better yet bookmark them in your favorite browser.

The Cheatsheet comes as a PDF or Excel:

- PDF A4 page
- Excel A4 page

2.7 Additional examples

The following examples are included in the SimPy source distribution.

2.7.1 Car

```
from SimPy.Simulation import Process, activate, hold, initialize, now, simulate


class Car(Process):
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield hold, self, 100.0
        print('%s %s %s' % (now(), self.name, 'Arrived'))
```

```
initialize()
c1 = Car('Car1', 2000)                  # a new car
activate(c1, c1.go(), at=6.0)     # activate at time 6.0
c2 = Car('Car2', 1600)                  # another new car
activate(c2, c2.go())                  # activate at time 0
simulate(until=200)
print('Current time is %s' % now())     # will print 106.0
```

Output:

```
0 Car2 Starting
6.0 Car1 Starting
100.0 Car2 Arrived
106.0 Car1 Arrived
Current time is 106.0
```

2.7.2 CarT

```
from SimPy.SimulationTrace import (Process, activate, initialize, hold, now,
                                   simulate)


class Car(Process):
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield hold, self, 100.0
        print('%s %s %s' % (now(), self.name, 'Arrived'))


initialize()
c1 = Car('Car1', 2000)                  # a new car
activate(c1, c1.go(), at=6.0)     # activate at time 6.0
c2 = Car('Car2', 1600)                  # another new car
activate(c2, c2.go())                  # activate at time 0
simulate(until=200)
print('Current time is %s' % now())     # will print 106.0
```

Output:

```
0 activate <Car1> at time: 6.0 prior: False
0 activate <Car2> at time: 0 prior: False
0 Car2 Starting
0 hold <Car2> delay: 100.0
6.0 Car1 Starting
6.0 hold <Car1> delay: 100.0
100.0 Car2 Arrived
100.0 <Car2> terminated
106.0 Car1 Arrived
106.0 <Car1> terminated
Current time is 106.0
```

2.7.3 Cars

```
from SimPy.Simulation import (Process, Resource, activate, initialize, hold,
                              now, release, request, simulate)


class Car(Process):
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield request, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Got a pump'))
        yield hold, self, 100.0
        yield release, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Leaving'))


gasstation = Resource(capacity=2, name='gasStation', unitName='pump')
initialize()
c1 = Car('Car1', 2000)
c2 = Car('Car2', 1600)
c3 = Car('Car3', 3000)
c4 = Car('Car4', 1600)
activate(c1, c1.go(), at=4.0)  # activate at time 4.0
activate(c2, c2.go())          # activate at time 0.0
activate(c3, c3.go(), at=3.0)  # activate at time 3.0
activate(c4, c4.go(), at=3.0)  # activate at time 2.0
simulate(until=300)
print('Current time is %s' % now())
```

Output:

```
0 Car2 Starting
0 Car2 Got a pump
3.0 Car3 Starting
3.0 Car3 Got a pump
3.0 Car4 Starting
4.0 Car1 Starting
100.0 Car2 Leaving
100.0 Car4 Got a pump
103.0 Car3 Leaving
103.0 Car1 Got a pump
200.0 Car4 Leaving
203.0 Car1 Leaving
Current time is 203.0
```

2.7.4 CarsT

```
from SimPy.SimulationTrace import (Process, Resource, activate, initialize,
                                   hold, now, release, request, simulate)


class Car(Process):
```

```
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield request, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Got a pump'))
        yield hold, self, 100.0
        yield release, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Leaving'))


gasstation = Resource(capacity=2, name='gasStation', unitName='pump')
initialize()
c1 = Car('Car1', 2000)
c2 = Car('Car2', 1600)
c3 = Car('Car3', 3000)
c4 = Car('Car4', 1600)
activate(c1, c1.go(), at=4.0)  # activate at time 4.0
activate(c2, c2.go())          # activate at time 0.0
activate(c3, c3.go(), at=3.0)  # activate at time 3.0
activate(c4, c4.go(), at=3.0)  # activate at time 2.0
simulate(until=300)
print('Current time is %s' % now())
```

Output:

```
0 activate <Car1> at time: 4.0 prior: False
0 activate <Car2> at time: 0 prior: False
0 activate <Car3> at time: 3.0 prior: False
0 activate <Car4> at time: 3.0 prior: False
0 Car2 Starting
0 request <Car2> <gasStation> priority: default
. . .waitQ: []
. . .activeQ: ['Car2']
0 Car2 Got a pump
0 hold <Car2> delay: 100.0
3.0 Car3 Starting
3.0 request <Car3> <gasStation> priority: default
. . .waitQ: []
. . .activeQ: ['Car2', 'Car3']
3.0 Car3 Got a pump
3.0 hold <Car3> delay: 100.0
3.0 Car4 Starting
3.0 request <Car4> <gasStation> priority: default
. . .waitQ: ['Car4']
. . .activeQ: ['Car2', 'Car3']
4.0 Car1 Starting
4.0 request <Car1> <gasStation> priority: default
. . .waitQ: ['Car4', 'Car1']
. . .activeQ: ['Car2', 'Car3']
100.0 reactivate <Car4> time: 100.0 prior: 1
100.0 release <Car2> <gasStation>
. . .waitQ: ['Car1']
. . .activeQ: ['Car3', 'Car4']
100.0 Car2 Leaving
100.0 <Car2> terminated
```

```
100.0 Car4 Got a pump
100.0 hold <Car4> delay: 100.0
103.0 reactivate <Car1> time: 103.0 prior: 1
103.0 release <Car3> <gasStation>
. . .waitQ: []
. . .activeQ: ['Car4', 'Car1']
103.0 Car3 Leaving
103.0 <Car3> terminated
103.0 Car1 Got a pump
103.0 hold <Car1> delay: 100.0
200.0 release <Car4> <gasStation>
. . .waitQ: []
. . .activeQ: ['Car1']
200.0 Car4 Leaving
200.0 <Car4> terminated
203.0 release <Car1> <gasStation>
. . .waitQ: []
. . .activeQ: []
203.0 Car1 Leaving
203.0 <Car1> terminated
Current time is 203.0
```

2.7.5 Carwash

```
from SimPy.Simulation import (Process, SimEvent, Store, activate, get,
                              initialize, hold, now, put, simulate, waitevent)
"""Carwash is master"""


class Carwash(Process):
    """Carwash is master"""

    def __init__(self, name):
        Process.__init__(self, name=name)

    def lifecycle(self):
        while True:
            yield get, self, waitingCars, 1
            carBeingWashed = self.got[0]
            yield hold, self, washtime
            carBeingWashed.doneSignal.signal(self.name)


class Car(Process):
    """Car is slave"""

    def __init__(self, name):
        Process.__init__(self, name=name)
        self.doneSignal = SimEvent()

    def lifecycle(self):
        yield put, self, waitingCars, [self]
        yield waitevent, self, self.doneSignal
        whichWash = self.doneSignal.signalparam
        print('%s car %s done by %s' % (now(), self.name, whichWash))
```

```
class CarGenerator(Process):
    def generate(self):
        i = 0
        while True:
            yield hold, self, 2
            c = Car('%d' % i)
            activate(c, c.lifecycle())
            i += 1


washtime = 5
initialize()

# put four cars into the queue of waiting cars
for j in range(1, 5):
    c = Car(name='%d' % -j)
    activate(c, c.lifecycle())

waitingCars = Store(capacity=40)
for i in range(2):
    cw = Carwash('Carwash %s' % i)
    activate(cw, cw.lifecycle())

cg = CarGenerator()
activate(cg, cg.generate())
simulate(until=30)
print('waitingCars %s' % [x.name for x in waitingCars.theBuffer])
```

Output:

```
5 car -1 done by Carwash 0
5 car -2 done by Carwash 1
10 car -3 done by Carwash 0
10 car -4 done by Carwash 1
15 car 0 done by Carwash 0
15 car 1 done by Carwash 1
20 car 2 done by Carwash 0
20 car 3 done by Carwash 1
25 car 4 done by Carwash 0
25 car 5 done by Carwash 1
30 car 6 done by Carwash 0
30 car 7 done by Carwash 1
waitingCars ['10', '11', '12', '13', '14']
```

2.7.6 Breakdown

```
from SimPy.Simulation import (Process, activate, hold, initialize, now,
                              reactivate, simulate)


class Bus(Process):

    def operate(self, repairduration, triplength):  # PEM
        tripleft = triplength                    # time needed to finish trip
        while tripleft > 0:
```

```
            yield hold, self, tripleft          # try to finish the trip
            if self.interrupted():              # if another breakdown occurs
                print('%s at %s' % (self.interruptCause.name, now()))
                tripleft = self.interruptLeft   # time to finish the trip
                self.interruptReset()           # end interrupt state
                reactivate(br, delay=repairduration)  # restart breakdown br
                yield hold, self, repairduration      # delay for repairs
                print('Bus repaired at %s' % now())
            else:
                break  # no more breakdowns, bus finished trip
        print('Bus has arrived at %s' % now())


class Breakdown(Process):
    def __init__(self, myBus):
        Process.__init__(self, name='Breakdown ' + myBus.name)
        self.bus = myBus

    def breakBus(self, interval):       # process execution method
        while True:
            yield hold, self, interval  # breakdown interarrivals
            if self.bus.terminated():
                break
            self.interrupt(self.bus)    # breakdown to myBus


initialize()
b = Bus('Bus')      # create a bus object
activate(b, b.operate(repairduration=20, triplength=1000))
br = Breakdown(b)   # create breakdown br to bus b
activate(br, br.breakBus(300))
simulate(until=4000)
print('SimPy: No more events at time %s' % now())
```

Output:

```
Breakdown Bus at 300
Bus repaired at 320
Breakdown Bus at 620
Bus repaired at 640
Breakdown Bus at 940
Bus repaired at 960
Bus has arrived at 1060
SimPy: No more events at time 1260
```

2.7.7 Diffpriority

```
from SimPy.Simulation import (PriorityQ, Process, Resource, activate,
                              initialize, hold, now, release, request,
                              simulate)


class Client(Process):
    def __init__(self, name):
        Process.__init__(self, name)
```

```
    def getserved(self, servtime, priority, myServer):
        print('%s requests 1 unit at t=%s' % (self.name, now()))
        yield request, self, myServer, priority
        yield hold, self, servtime
        yield release, self, myServer
        print('%s done at t=%s' % (self.name, now()))


initialize()
# create the *server* Resource object
server = Resource(capacity=1, qType=PriorityQ, preemptable=1)
# create some Client process objects
c1 = Client(name='c1')
c2 = Client(name='c2')
activate(c1, c1.getserved(servtime=100, priority=1, myServer=server), at=0)
activate(c2, c2.getserved(servtime=100, priority=9, myServer=server), at=50)
simulate(until=500)
```

Output:

```
c1 requests 1 unit at t=0
c2 requests 1 unit at t=50
c2 done at t=150
c1 done at t=200
```

2.7.8 Firework

```
from SimPy.Simulation import Process, activate, initialize, hold, now, simulate


class Firework(Process):

    def execute(self):
        print('%s firework launched' % now())
        yield hold, self, 10.0  # wait 10.0 time units
        for i in range(10):
            yield hold, self, 1.0
            print('%s tick' % now())
        yield hold, self, 10.0  # wait another 10.0 time units
        print('%s Boom!!' % now())


initialize()
f = Firework()  # create a Firework object, and
# activate it (with some default parameters)
activate(f, f.execute(), at=0.0)
simulate(until=100)
```

Output:

```
0 firework launched
11.0 tick
12.0 tick
13.0 tick
14.0 tick
15.0 tick
```

```
16.0 tick
17.0 tick
18.0 tick
19.0 tick
20.0 tick
30.0 Boom!!
```

2.7.9 Levelinventory

```
from random import normalvariate, seed
from SimPy.Simulation import (Level, Process, activate, get, initialize, hold,
                              now, put, simulate)


class Deliver(Process):
    def deliver(self):  # an "offeror" PEM
        while True:
            lead = 10.0      # time between refills
            delivery = 10.0  # amount in each refill
            yield put, self, stock, delivery
            print('at %6.4f, add %6.4f units, now amount = %6.4f' %
                  (now(), delivery, stock.amount))
            yield hold, self, lead


class Demand(Process):
    stockout = 0.0     # initialize initial stockout amount

    def demand(self):  # a "requester" PEM
        day = 1.0      # set time-step to one day
        while True:
            yield hold, self, day
            dd = normalvariate(1.20, 0.20)  # today's random demand
            ds = dd - stock.amount
            # excess of demand over current stock amount
            if dd > stock.amount:     # can't supply requested amount
                yield get, self, stock, stock.amount
                # supply all available amount
                self.stockout += ds
                # add unsupplied demand to self.stockout
                print('day %6.4f, demand = %6.4f, shortfall = %6.4f' %
                      (now(), dd, -ds))
            else:  # can supply requested amount
                yield get, self, stock, dd
                print('day %6.4f, supplied %6.4f, now amount = %6.4f' %
                      (now(), dd, stock.amount))


stock = Level(monitored=True)     # 'unbounded' capacity and other defaults

seed(99999)
initialize()

offeror = Deliver()
activate(offeror, offeror.deliver())
requester = Demand()
```

```
activate(requester, requester.demand())

simulate(until=49.9)

result = (stock.bufferMon.mean(), requester.stockout)
print('')
print('Summary of results through end of day %6.4f:' % int(now()))
print('average stock = %6.4f, cumulative stockout = %6.4f' % result)
```

Output:

```
at 0.0000, add 10.0000 units, now amount = 10.0000
day 1.0000, supplied 1.0115, now amount = 8.9885
day 2.0000, supplied 1.1377, now amount = 7.8508
day 3.0000, supplied 0.7743, now amount = 7.0766
day 4.0000, supplied 1.4056, now amount = 5.6709
day 5.0000, supplied 0.9736, now amount = 4.6973
day 6.0000, supplied 1.1061, now amount = 3.5912
day 7.0000, supplied 1.4067, now amount = 2.1845
day 8.0000, supplied 1.2884, now amount = 0.8961
day 9.0000, demand = 1.4426, shortfall = -0.5465
at 10.0000, add 10.0000 units, now amount = 10.0000
day 10.0000, supplied 1.5850, now amount = 8.4150
day 11.0000, supplied 0.7974, now amount = 7.6176
day 12.0000, supplied 0.8854, now amount = 6.7323
day 13.0000, supplied 1.4893, now amount = 5.2430
day 14.0000, supplied 1.0522, now amount = 4.1908
day 15.0000, supplied 1.0492, now amount = 3.1416
day 16.0000, supplied 1.2013, now amount = 1.9403
day 17.0000, supplied 1.4026, now amount = 0.5377
day 18.0000, demand = 1.0874, shortfall = -0.5497
day 19.0000, demand = 1.3043, shortfall = -1.3043
at 20.0000, add 10.0000 units, now amount = 10.0000
day 20.0000, supplied 1.2737, now amount = 8.7263
day 21.0000, supplied 0.9550, now amount = 7.7713
day 22.0000, supplied 1.3646, now amount = 6.4067
day 23.0000, supplied 0.8629, now amount = 5.5438
day 24.0000, supplied 1.2893, now amount = 4.2545
day 25.0000, supplied 1.1897, now amount = 3.0648
day 26.0000, supplied 0.9708, now amount = 2.0940
day 27.0000, supplied 1.1397, now amount = 0.9543
day 28.0000, demand = 1.1286, shortfall = -0.1743
day 29.0000, demand = 0.8891, shortfall = -0.8891
at 30.0000, add 10.0000 units, now amount = 10.0000
day 30.0000, supplied 0.9496, now amount = 9.0504
day 31.0000, supplied 1.1515, now amount = 7.8988
day 32.0000, supplied 1.3724, now amount = 6.5264
day 33.0000, supplied 0.6977, now amount = 5.8287
day 34.0000, supplied 1.2291, now amount = 4.5996
day 35.0000, supplied 1.3502, now amount = 3.2494
day 36.0000, supplied 1.0396, now amount = 2.2098
day 37.0000, supplied 1.3442, now amount = 0.8656
day 38.0000, demand = 1.0919, shortfall = -0.2263
day 39.0000, demand = 1.4077, shortfall = -1.4077
at 40.0000, add 10.0000 units, now amount = 10.0000
day 40.0000, supplied 0.9053, now amount = 9.0947
day 41.0000, supplied 1.2982, now amount = 7.7965
day 42.0000, supplied 1.3458, now amount = 6.4508
```

```
day 43.0000, supplied 1.2357, now amount = 5.2151
day 44.0000, supplied 1.2822, now amount = 3.9329
day 45.0000, supplied 1.1809, now amount = 2.7519
day 46.0000, supplied 1.0891, now amount = 1.6629
day 47.0000, supplied 1.1926, now amount = 0.4702
day 48.0000, demand = 1.3927, shortfall = -0.9224
day 49.0000, demand = 1.2690, shortfall = -1.2690

Summary of results through end of day 49.0000:
average stock = 4.4581, cumulative stockout = 7.2894
```

2.7.10 Message

```
from SimPy.Simulation import Process, activate, initialize, hold, now, simulate


class Message(Process):
    """A simple Process"""

    def __init__(self, i, len):
        Process.__init__(self, name='Message' + str(i))
        self.i = i
        self.len = len

    def go(self):
        print('%s %s %s' % (now(), self.i, 'Starting'))
        yield hold, self, 100.0
        print('%s %s %s' % (now(), self.i, 'Arrived'))


initialize()
p1 = Message(1, 203)    # new message
activate(p1, p1.go())  # activate it
p2 = Message(2, 33)
activate(p2, p2.go(), at=6.0)
simulate(until=200)
print('Current time is %s' % now())  # will print 106.0
```

Output:

```
0 1 Starting
6.0 2 Starting
100.0 1 Arrived
106.0 2 Arrived
Current time is 106.0
```

2.7.11 Monitor

```
from SimPy.Simulation import Monitor
from random import expovariate

m = Monitor()         # define the Monitor object, m

for i in range(1000):     # make the observations
```

```
    y = expovariate(0.1)
    m.observe(y)

# set up and return the completed histogram
h = m.histogram(low=0.0, high=20, nbins=30)
```

2.7.12 Resource

```
from SimPy.Simulation import (Process, Resource, activate, initialize, hold,
                              now, release, request, simulate)


class Client(Process):
    inClients = []     # list the clients in order by their requests
    outClients = []    # list the clients in order by completion of service

    def __init__(self, name):
        Process.__init__(self, name)

    def getserved(self, servtime, priority, myServer):
        Client.inClients.append(self.name)
        print('%s requests 1 unit at t = %s' % (self.name, now()))
        # request use of a resource unit
        yield request, self, myServer, priority
        yield hold, self, servtime
        # release the resource
        yield release, self, myServer
        print('%s done at t = %s' % (self.name, now()))
        Client.outClients.append(self.name)


initialize()

# the next line creates the ``server`` Resource object
server = Resource(capacity=2)  # server defaults to qType==FIFO

# the next lines create some Client process objects
c1, c2 = Client(name='c1'), Client(name='c2')
c3, c4 = Client(name='c3'), Client(name='c4')
c5, c6 = Client(name='c5'), Client(name='c6')

# in the next lines each client requests
# one of the ``server``'s Resource units
activate(c1, c1.getserved(servtime=100, priority=1, myServer=server))
activate(c2, c2.getserved(servtime=100, priority=2, myServer=server))
activate(c3, c3.getserved(servtime=100, priority=3, myServer=server))
activate(c4, c4.getserved(servtime=100, priority=4, myServer=server))
activate(c5, c5.getserved(servtime=100, priority=5, myServer=server))
activate(c6, c6.getserved(servtime=100, priority=6, myServer=server))

simulate(until=500)

print('Request order: %s' % Client.inClients)
print('Service order: %s' % Client.outClients)
```

Output:

```
c1 requests 1 unit at t = 0
c2 requests 1 unit at t = 0
c3 requests 1 unit at t = 0
c4 requests 1 unit at t = 0
c5 requests 1 unit at t = 0
c6 requests 1 unit at t = 0
c1 done at t = 100
c2 done at t = 100
c3 done at t = 200
c4 done at t = 200
c5 done at t = 300
c6 done at t = 300
Request order: ['c1', 'c2', 'c3', 'c4', 'c5', 'c6']
Service order: ['c1', 'c2', 'c3', 'c4', 'c5', 'c6']
```

2.7.13 Resource monitor

```
from math import sqrt
from SimPy.Simulation import (Monitor, Process, Resource, activate, initialize,
                              hold, now, release, request, simulate)


class Client(Process):
    inClients = []
    outClients = []

    def __init__(self, name):
        Process.__init__(self, name)

    def getserved(self, servtime, myServer):
        print('%s requests 1 unit at t = %s' % (self.name, now()))
        yield request, self, myServer
        yield hold, self, servtime
        yield release, self, myServer
        print('%s done at t = %s' % (self.name, now()))


initialize()

server = Resource(capacity=1, monitored=True, monitorType=Monitor)

c1, c2 = Client(name='c1'), Client(name='c2')
c3, c4 = Client(name='c3'), Client(name='c4')

activate(c1, c1.getserved(servtime=100, myServer=server))
activate(c2, c2.getserved(servtime=100, myServer=server))
activate(c3, c3.getserved(servtime=100, myServer=server))
activate(c4, c4.getserved(servtime=100, myServer=server))

simulate(until=500)

print('')
print('(TimeAverage no. waiting: %s' % server.waitMon.timeAverage())
print('(Number) Average no. waiting: %.4f' % server.waitMon.mean())
print('(Number) Var of no. waiting: %.4f' % server.waitMon.var())
print('(Number) SD of no. waiting: %.4f' % sqrt(server.waitMon.var()))
```

```
print('(TimeAverage no. in service: %s' % server.actMon.timeAverage())
print('(Number) Average no. in service: %.4f' % server.actMon.mean())
print('(Number) Var of no. in service: %.4f' % server.actMon.var())
print('(Number) SD of no. in service: %.4f' % sqrt(server.actMon.var()))
print('=' * 40)
print('Time history for the "server" waitQ:')
print('[time, waitQ]')
for item in server.waitMon:
    print(item)
print('=' * 40)
print('Time history for the "server" activeQ:')
print('[time, activeQ]')
for item in server.actMon:
    print(item)
```

Output:

```
c1 requests 1 unit at t = 0
c2 requests 1 unit at t = 0
c3 requests 1 unit at t = 0
c4 requests 1 unit at t = 0
c1 done at t = 100
c2 done at t = 200
c3 done at t = 300
c4 done at t = 400

(TimeAverage no. waiting: 1.5
(Number) Average no. waiting: 1.2857
(Number) Var of no. waiting: 1.0612
(Number) SD of no. waiting: 1.0302
(TimeAverage no. in service: 1.0
(Number) Average no. in service: 0.4444
(Number) Var of no. in service: 0.2469
(Number) SD of no. in service: 0.4969
========================================
Time history for the "server" waitQ:
[time, waitQ]
[0, 0]
[0, 1]
[0, 2]
[0, 3]
[100, 2]
[200, 1]
[300, 0]
========================================
Time history for the "server" activeQ:
[time, activeQ]
[0, 0]
[0, 1]
[100, 0]
[100, 1]
[200, 0]
[200, 1]
[300, 0]
[300, 1]
[400, 0]
```

2.7.14 Romulans

```
from SimPy.Simulation import (Process, initialize, activate, simulate,
                              hold, now, waituntil, stopSimulation)
import random


class Player(Process):

    def __init__(self, lives=1, name='ImaTarget'):
        Process.__init__(self, name)
        self.lives = lives
        # provide Player objects with a "damage" property
        self.damage = 0

    def life(self):
        self.message = 'Drat! Some %s survived Federation attack!' % \
            (target.name)

        def killed():     # function testing for "damage > 5"
            return self.damage > 5

        while True:
            yield waituntil, self, killed
            self.lives -= 1
            self.damage = 0
            if self.lives == 0:
                self.message = '%s wiped out by Federation at \
                time %s!' % (target.name, now())
                stopSimulation()


class Federation(Process):

    def fight(self):                # simulate Federation operations
        print('Three %s attempting to escape!' % (target.name))
        while True:
            if random.randint(0, 10) < 2:  # check for hit on player
                target.damage += 1         # hit! increment damage to player
                if target.damage <= 5:     # target survives
                    print('Ha! %s hit! Damage= %i' %
                          (target.name, target.damage))
                else:
                    if (target.lives - 1) == 0:
                        print('No more %s left!' % (target.name))
                    else:
                        print('Now only %i %s left!' % (target.lives - 1,
                                                        target.name))

            yield hold, self, 1


initialize()
gameOver = 100
# create a Player object named "Romulans"
target = Player(lives=3, name='Romulans')
activate(target, target.life())
# create a Federation object
```

```
shooter = Federation()
activate(shooter, shooter.fight())
simulate(until=gameOver)
print(target.message)
```

Example output, varies:

```
Three Romulans attempting to escape!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Ha! Romulans hit! Damage= 3
Ha! Romulans hit! Damage= 4
Ha! Romulans hit! Damage= 5
Now only 2 Romulans left!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Ha! Romulans hit! Damage= 3
Ha! Romulans hit! Damage= 4
Ha! Romulans hit! Damage= 5
Now only 1 Romulans left!
Ha! Romulans hit! Damage= 1
Ha! Romulans hit! Damage= 2
Drat! Some Romulans survived Federation attack!
```

2.7.15 Shopping

```
from SimPy.Simulation import Process, activate, hold, initialize, simulate


class Customer(Process):
    def buy(self, budget=0):
        print('Here I am at the shops %s' % self.name)
        t = 5.0
        for i in range(4):
            yield hold, self, t
            # executed 4 times at intervals of t time units
            print('I just bought something %s' % self.name)
            budget -= 10.00
        print('All I have left is %s I am going home %s' % (budget, self.name))


initialize()

# create a customer named "Evelyn",
C = Customer(name='Evelyn')

# and activate her with a budget of 100
activate(C, C.buy(budget=100), at=10.0)

simulate(until=100.0)
```

Output:

```
Here I am at the shops Evelyn
I just bought something Evelyn
I just bought something Evelyn
```

```
I just bought something Evelyn
I just bought something Evelyn
All I have left is 60.0 I am going home Evelyn
```

2.7.16 Storewidget

```
from SimPy.Simulation import (Lister, Process, Store, activate, get, hold,
                              initialize, now, put, simulate)


class ProducerD(Process):
    def __init__(self):
        Process.__init__(self)

    def produce(self):  # the ProducerD PEM
        while True:
            yield put, self, buf, [Widget(9), Widget(7)]
            yield hold, self, 10


class ConsumerD(Process):
    def __init__(self):
        Process.__init__(self)

    def consume(self):  # the ConsumerD PEM
        while True:
            toGet = 3
            yield get, self, buf, toGet
            assert len(self.got) == toGet
            print('%s Get widget weights %s' % (now(),
                                                [x.weight for x in self.got]))
            yield hold, self, 11


class Widget(Lister):
    def __init__(self, weight=0):
        self.weight = weight


widgbuf = []
for i in range(10):
    widgbuf.append(Widget(5))

initialize()

buf = Store(capacity=11, initialBuffered=widgbuf, monitored=True)

for i in range(3):  # define and activate 3 producer objects
    p = ProducerD()
    activate(p, p.produce())

for i in range(3):  # define and activate 3 consumer objects
    c = ConsumerD()
    activate(c, c.consume())

simulate(until=50)
```

```
print('LenBuffer: %s' % buf.bufferMon)  # length of buffer
print('getQ: %s' % buf.getQMon)         # length of getQ
print('putQ %s' % buf.putQMon)          # length of putQ
```

Output:

```
0 Get widget weights [5, 5, 5]
0 Get widget weights [5, 5, 5]
0 Get widget weights [5, 5, 5]
11 Get widget weights [5, 9, 7]
11 Get widget weights [9, 7, 9]
11 Get widget weights [7, 9, 7]
22 Get widget weights [9, 7, 9]
22 Get widget weights [7, 9, 7]
22 Get widget weights [9, 7, 9]
33 Get widget weights [7, 9, 7]
33 Get widget weights [9, 7, 9]
40 Get widget weights [7, 9, 7]
44 Get widget weights [9, 7, 9]
50 Get widget weights [7, 9, 7]
LenBuffer: [[0, 10], [0, 7], [0, 9], [0, 11], [0, 8], [0, 10], [0, 7], [10, 9], [10,␣
↪11], [11, 8], [11, 10], [11, 7], [11, 4], [20, 6], [20, 8], [21, 10], [22, 7], [22,␣
↪4], [22, 1], [30, 3], [30, 5], [31, 7], [33, 4], [33, 1], [40, 3], [40, 0], [40, 2],
↪ [41, 4], [44, 1], [50, 3], [50, 0], [50, 2]]
getQ: [[0, 0], [33, 1], [40, 0], [44, 1], [50, 0]]
putQ [[0, 0], [0, 1], [0, 2], [0, 3], [0, 2], [0, 1], [0, 0], [10, 1], [11, 0]]
```

2.7.17 Tally

```
from SimPy.Simulation import Tally
import random as r

t = Tally(name="myTally", ylab="wait time (sec)")
t.setHistogram(low=0.0, high=1.0, nbins=10)
for i in range(100000):
    t.observe(y=r.random())
print(t.printHistogram(fmt="%6.4f"))
```

Output:

```
Histogram for myTally:
Number of observations: 100000
          wait time (sec) < 0.0000:      0 (cum:      0/  0.0%)
0.0000 <= wait time (sec) < 0.1000:  10135 (cum:  10135/ 10.1%)
0.1000 <= wait time (sec) < 0.2000:   9973 (cum:  20108/ 20.1%)
0.2000 <= wait time (sec) < 0.3000:  10169 (cum:  30277/ 30.3%)
0.3000 <= wait time (sec) < 0.4000:  10020 (cum:  40297/ 40.3%)
0.4000 <= wait time (sec) < 0.5000:  10126 (cum:  50423/ 50.4%)
0.5000 <= wait time (sec) < 0.6000:   9866 (cum:  60289/ 60.3%)
0.6000 <= wait time (sec) < 0.7000:   9910 (cum:  70199/ 70.2%)
0.7000 <= wait time (sec) < 0.8000:   9990 (cum:  80189/ 80.2%)
0.8000 <= wait time (sec) < 0.9000:   9852 (cum:  90041/ 90.0%)
0.9000 <= wait time (sec) < 1.0000:   9959 (cum: 100000/100.0%)
1.0000 <= wait time (sec)          :      0 (cum: 100000/100.0%)
```

Second tally example:

```
from SimPy.Simulation import Tally
from random import expovariate

r = Tally('Tally')  # define a tally object, r
r.setHistogram(name='exponential',
               low=0.0, high=20.0, nbins=30)  # set before observations

for i in range(1000):    # make the observations
    y = expovariate(0.1)
    r.observe(y)

h = r.getHistogram()     # return the completed histogram
print(h)
```

2.8 SimPy Classic - Examples

These are the examples that ship with the SimPy source.

These pages are to index the examples that ship with SimPy and give users another way to find them.

Contents:

2.8.1  LIST OF MODELS using SimPy Classic

SimPy version 2.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7 or later and 3.x except for the GUI ones

These models are examples of SimPy use written by several authors and usually developed for other purposes, such as teaching and consulting. They are in a variety of styles.

Most models are given in two versions, one with the procedural SimPy API and the other (identified by an "_OO" appended to the program name) with the Object Oriented API introduced in SimPy 2.0.

All of these examples come with SimPY in the docs/examples directory.

NOTE: The SimGUI examples do not work for Python 3 as the SimGUI library has not been ported to Python 3.

New Program Structure

M/M/C Queue: MCC.py, MCC_OO.py

M/M/C (multiple server queue model). This demonstrates both the multiple capacity Resource class and the observe method of the Monitor class. Random arrivals, exponential service-times. (TV)

Jobs arrive at random into a c-server queue with exponential service-time distribution. Simulate to determine the average number in the system and the average time jobs spend in the system.

```
"""MMC.py

An M/M/c queue

Jobs arrive at random into a c-server queue with
exponential service-time distribution. Simulate to
determine the average  number in the system and
the average time jobs spend in the system.

- c = Number of servers = 3
- rate = Arrival rate = 2.0
- stime = mean service time = 1.0

"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Generator(Process):
    """ generates Jobs at random """

    def execute(self, maxNumber, rate, stime):
        ''' generate Jobs at exponential intervals '''
        for i in range(maxNumber):
            L = Job("Job {0} ".format(i))
            activate(L, L.execute(stime), delay=0)
            yield hold, self, expovariate(rate)


class Job(Process):
    ''' Jobs request a gatekeeper and hold
        it for an exponential time '''

    NoInSystem = 0

    def execute(self, stime):
        arrTime = now()
        self.trace("Hello World")
        Job.NoInSystem += 1
        m.observe(Job.NoInSystem)
        yield request, self, server
        self.trace("At last    ")
        t = expovariate(1.0 / stime)
        msT.observe(t)
        yield hold, self, t
        yield release, self, server
        Job.NoInSystem -= 1
        m.observe(Job.NoInSystem)
        mT.observe(now() - arrTime)
        self.trace("Geronimo   ")

    def trace(self, message):
        FMT = "{0:7.4f} {1:6} {2:10} ({3:2d})"
        if TRACING:
            print(FMT.format(now(), self.name, message, Job.NoInSystem))
# Experiment data ------------------------
```

```
TRACING = False
c = 3  # number of servers in M/M/c system
stime = 1.0  # mean service time
rate = 2.0  # mean arrival rate
maxNumber = 1000
m = Monitor()  # monitor for the number of jobs
mT = Monitor()  # monitor for the time in system
msT = Monitor()  # monitor for the generated service times

seed(333555777)  # seed for random numbers


server = Resource(capacity=c, name='Gatekeeper')


# Model/Experiment ------------------------------

initialize()
g = Generator('gen')
activate(g, g.execute(maxNumber=maxNumber,
                      rate=rate, stime=stime))
m.observe(0)  # number in system is 0 at the start
simulate(until=3000.0)


# Analysis/output --------------------------

print('MMC')
print("{0:2d} servers, {1:6.4f} arrival rate, "
      "{2:6.4f} mean service time".format(c, rate, stime))
print("Average number in the system is {0:6.4f}".format(m.timeAverage()))
print("Average time in the system is   {0:6.4f}".format(mT.mean()))
print("Actual average service-time is  {0:6.4f}".format(msT.mean()))
```

```
MMC
 3 servers, 2.0000 arrival rate, 1.0000 mean service time
Average number in the system is 2.8786
Average time in the system is   1.4350
Actual average service-time is  0.9837
```

```
"""MMC_OO.py

An M/M/c queue

Jobs arrive at random into a c-server queue with
exponential service-time distribution. Simulate to
determine the average  number in the system and
the average time jobs spend in the system.

- c = Number of servers = 3
- rate = Arrival rate = 2.0
- stime = mean service time = 1.0

"""
from SimPy.Simulation import *
```

```
from random import expovariate, seed

# Model components ------------------------


class Generator(Process):
    """ generates Jobs at random """

    def execute(self, maxNumber, rate, stime):
        ''' generate Jobs at exponential intervals '''
        for i in range(maxNumber):
            L = Job("Job {0}".format(i), sim=self.sim)
            self.sim.activate(L, L.execute(stime), delay=0)
            yield hold, self, expovariate(rate)


class Job(Process):
    ''' Jobs request a gatekeeper and hold
        it for an exponential time '''

    NoInSystem = 0

    def execute(self, stime):
        arrTime = self.sim.now()
        self.trace("Hello World")
        Job.NoInSystem += 1
        self.sim.m.observe(Job.NoInSystem)
        yield request, self, self.sim.server
        self.trace("At last    ")
        t = expovariate(1.0 / stime)
        self.sim.msT.observe(t)
        yield hold, self, t
        yield release, self, self.sim.server
        Job.NoInSystem -= 1
        self.sim.m.observe(Job.NoInSystem)
        self.sim.mT.observe(self.sim.now() - arrTime)
        self.trace("Geronimo   ")

    def trace(self, message):
        FMT = "{0:7.4f} {1:6} {2:10} {(3:2d)}"
        if TRACING:
            print(FMT.format(self.sim.now(), self.name,
                             message, Job.NoInSystem))

# Experiment data -------------------------


TRACING = False
c = 3  # number of servers in M/M/c system
stime = 1.0  # mean service time
rate = 2.0  # mean arrival rate
maxNumber = 1000
seed(333555777)  # seed for random numbers

# Model -----------------------------------


class MMCmodel(Simulation):
```

```
    def run(self):
        self.initialize()
        self.m = Monitor(sim=self)  # monitor for the number of jobs
        self.mT = Monitor(sim=self)  # monitor for the time in system
        self.msT = Monitor(sim=self)  # monitor for the generated service times
        self.server = Resource(capacity=c, name='Gatekeeper', sim=self)
        g = Generator(name='gen', sim=self)
        self.activate(g, g.execute(maxNumber=maxNumber,
                                   rate=rate, stime=stime))
        self.m.observe(0)  # number in system is 0 at the start
        self.simulate(until=3000.0)


# Experiment ------------------------------
model = MMCmodel()
model.run()

# Analysis/output -------------------------

print('MMC')
print("{0:2d} servers, {1:6.4f} arrival rate, "
      "{2:6.4f} mean service time".format(c, rate, stime))
print("Average number in the system is {0:6.4f}".format(model.m.timeAverage()))
print("Average time in the system is   {0:6.4f}".format(model.mT.mean()))
print("Actual average service-time is  {0:6.4f}".format(model.msT.mean()))
```

```
MMC
 3 servers, 2.0000 arrival rate, 1.0000 mean service time
Average number in the system is 2.8786
Average time in the system is   1.4350
Actual average service-time is  0.9837
```

BCC: bcc.py, bcc_OO.py

Determine the probability of rejection of random arrivals to a 2-server system with different service-time distributions. No queues allowed, blocked customers are rejected (BCC). Distributions are Erlang, exponential, and hyperexponential. The theoretical probability is also calculated. (TV)

```
""" bcc.py

  Queue with blocked customers cleared
  Jobs (e.g messages) arrive randomly at rate 1.0 per minute at a
  2-server system.  Mean service time is 0.75 minutes and the
  service-time distribution is (1) exponential, (2) Erlang-5, or (3)
  hyperexponential with p=1/8,m1=2.0, and m2=4/7. However no
  queue is allowed; a job arriving when all the servers are busy is
  rejected.

  Develop and run a simulation program to estimate the probability of
  rejection (which, in steady-state, is the same as p(c)) Measure
  and compare the probability for each service time distribution.
  Though you should test the program with a trace, running just a few
  jobs, the final runs should be of 10000 jobs without a trace. Stop
  the simulation when 10000 jobs have been generated.
"""
```

```
from SimPy.Simulation import *
from random import seed, Random, expovariate

# Model components ------------------------

dist = ""


def bcc(lam, mu, s):
    """ bcc - blocked customers cleared model

    - returns p[i], i = 0,1,..s.
    - ps = p[s] = prob of blocking
    - lameff = effective arrival rate = lam*(1-ps)

    See Winston 22.11 for Blocked Customers Cleared Model (Erlang B formula)
    """
    rho = lam / mu
    n = range(s + 1)
    p = [0] * (s + 1)
    p[0] = 1
    sump = 1.0
    for i in n[1:]:
        p[i] = (rho / i) * p[i - 1]
        sump = sump + p[i]
    p0 = 1.0 / sump
    for i in n:
        p[i] = p[i] * p0
    p0 = p[0]
    ps = p[s]
    lameff = lam * (1 - ps)
    L = rho * (1 - ps)
    return {'lambda': lam, 'mu': mu, 's': s,
            'p0': p0, 'p[i]': p, 'ps': ps, 'L': L}


def ErlangVariate(mean, K):
    """ Erlang random variate

    mean = mean
    K = shape parameter
    g = rv to be used
    """
    sum = 0.0
    mu = K / mean
    for i in range(K):
        sum += expovariate(mu)
    return (sum)


def HyperVariate(p, m1, m2):
    """ Hyperexponential random variate

    p = prob of branch 1
    m1 = mean of exponential, branch 1
    m2 = mean of exponential, branch 2
    g = rv to be used
    """
```

```
    if random() < p:
        return expovariate(1.0 / m1)
    else:
        return expovariate(1.0 / m2)


def testHyperVariate():
    """ tests the HyerVariate rv generator"""
    ERR = 0
    x = (1.0981, 1.45546, 5.7470156)
    p = 0.0, 1.0, 0.5
    g = Random(1113355)
    for i in range(3):
        x1 = HyperVariate(p[i], 1.0, 10.0, g)
        # print(p[i], x1)
        assert abs(x1 - x[i]) < 0.001, 'HyperVariate error'


def erlangB(rho, c):
    """ Erlang's B formula for probabilities in no-queue

    Returns p[n] list
    see also SPlus and R version in que.q mmcK
    que.py has bcc.
    """
    n = range(c + 1)
    pn = list(range(c + 1))
    term = 1
    pn[0] = 1
    sum = 1
    term = 1.0
    i = 1
    while i < (c + 1):
        term *= rho / i
        pn[i] = term
        sum += pn[i]
        i += 1
    for i in n:
        pn[i] = pn[i] / sum
    return(pn)


class JobGen(Process):
    """ generates a sequence of Jobs
    """

    def execute(self, JobRate, MaxJob, mu):
        global NoInService, Busy
        for i in range(MaxJob):
            j = Job()
            activate(j, j.execute(i, mu), delay=0.0)
            t = expovariate(JobRate)
            MT.tally(t)
            yield hold, self, t
        self.trace("Job generator finished")

    def trace(self, message):
        if JobGenTRACING:
```

```
            print("{0:8.4f} \t{1}".format(now(), message))


class Job(Process):
    """ Jobs that are either accepted or rejected
    """

    def execute(self, i, mu):
        """ Job execution, only if accepted"""
        global NoInService, Busy, dist, NoRejected
        if NoInService < c:
            self.trace("Job %2d accepted b=%1d" % (i, Busy))
            NoInService += 1
            if NoInService == c:
                Busy = 1
                try:
                    BM.accum(Busy, now())
                except:
                    "accum error BM=", BM
            # yield hold,self,Job.g.expovariate(self.mu);
            # dist= "Exponential"
            yield hold, self, ErlangVariate(1.0 / mu, 5)
            dist = "Erlang     "
            # yield hold,self,HyperVariate(1.0/8,m1=2.0,m2=4.0/7,g=Job.g);
            # dist= "HyperExpon "
            NoInService -= 1
            Busy = 0
            BM.accum(Busy, now())
            self.trace("Job %2d leaving b=%1d" % (i, Busy))
        else:
            self.trace("Job %2d REJECT  b=%1d" % (i, Busy))
            NoRejected += 1

    def trace(self, message):
        if JobTRACING:
            print("{0:8.4f} \t{1}".format(now(), message))


# Experiment data -------------------------
c = 2
lam = 1.0  # per minute
mu = 1.0 / 0.75  # per minute
p = 1.0 / 8
m1 = 2.0
m2 = 4.0 / 7.0
K = 5
rho = lam / mu

NoRejected = 0
NoInService = 0
Busy = 0

JobRate = lam
JobMax = 10000

JobTRACING = 0
JobGenTRACING = 0
```

```
# Model/Experiment ------------------------------

seed(111333)
BM = Monitor()
MT = Monitor()

initialize()
jbg = JobGen()
activate(jbg, jbg.execute(1.0, JobMax, mu), 0.0)
simulate(until=20000.0)

# Analysis/output -------------------------

print('bcc')
print("time at the end = {0}".format(now()))
print("now = {0}\tstartTime = {1}".format(now(), BM.startTime))
print("No Rejected = {0:d}, ratio= {1}".format(
    NoRejected, (1.0 * NoRejected) / JobMax))
print("Busy proportion ({0}) = {1:8.6f}".format(dist, BM.timeAverage()))
print("Erlang pc (th)                = {0:8.6f}".format(erlangB(rho, c)[c]))
```

```
bcc
time at the end = 10085.751051963694
now = 10085.751051963694        startTime = 0.0
No Rejected = 1374, ratio= 0.1374
Busy proportion (Erlang     ) = 0.139016
Erlang pc (th)                = 0.138462
```

```
""" bcc_OO.py

  Queue with blocked customers cleared
  Jobs (e.g messages) arrive randomly at rate 1.0 per minute at a
  2-server system.  Mean service time is 0.75 minutes and the
  service-time distribution is (1) exponential, (2) Erlang-5, or (3)
  hyperexponential with p=1/8,m1=2.0, and m2=4/7. However no
  queue is allowed; a job arriving when all the servers are busy is
  rejected.

  Develop and run a simulation program to estimate the probability of
  rejection (which, in steady-state, is the same as p(c)) Measure
  and compare the probability for each service time distribution.
  Though you should test the program with a trace, running just a few
  jobs, the final runs should be of 10000 jobs without a trace. Stop
  the simulation when 10000 jobs have been generated.
"""

from SimPy.Simulation import *
from random import seed, Random, expovariate

# Model components ------------------------

dist = ""


def bcc(lam, mu, s):
    """ bcc - blocked customers cleared model
```

```
    - returns p[i], i = 0,1,..s.
    - ps = p[s] = prob of blocking
    - lameff = effective arrival rate = lam*(1-ps)

    See Winston 22.11 for Blocked Customers Cleared Model (Erlang B formula)
    """
    rho = lam / mu
    n = range(s + 1)
    p = [0] * (s + 1)
    p[0] = 1
    sump = 1.0
    for i in n[1:]:
        p[i] = (rho / i) * p[i - 1]
        sump = sump + p[i]
    p0 = 1.0 / sump
    for i in n:
        p[i] = p[i] * p0
    p0 = p[0]
    ps = p[s]
    lameff = lam * (1 - ps)
    L = rho * (1 - ps)
    return {'lambda': lam, 'mu': mu, 's': s,
            'p0': p0, 'p[i]': p, 'ps': ps, 'L': L}


def ErlangVariate(mean, K):
    """ Erlang random variate

    mean = mean
    K = shape parameter
    g = rv to be used
    """
    sum = 0.0
    mu = K / mean
    for i in range(K):
        sum += expovariate(mu)
    return (sum)


def HyperVariate(p, m1, m2):
    """ Hyperexponential random variate

    p = prob of branch 1
    m1 = mean of exponential, branch 1
    m2 = mean of exponential, branch 2
    g = rv to be used
    """
    if random() < p:
        return expovariate(1.0 / m1)
    else:
        return expovariate(1.0 / m2)


def testHyperVariate():
    """ tests the HyerVariate rv generator"""
    ERR = 0
    x = (1.0981, 1.45546, 5.7470156)
```

```
    p = 0.0, 1.0, 0.5
    g = Random(1113355)
    for i in range(3):
        x1 = HyperVariate(p[i], 1.0, 10.0, g)
        # print(p[i], x1)
        assert abs(x1 - x[i]) < 0.001, 'HyperVariate error'


def erlangB(rho, c):
    """ Erlang's B formula for probabilities in no-queue

    Returns p[n] list
    see also SPlus and R version in que.q mmcK
    que.py has bcc.
    """
    n = range(c + 1)
    pn = list(range(c + 1))
    term = 1
    pn[0] = 1
    sum = 1
    term = 1.0
    i = 1
    while i < (c + 1):
        term *= rho / i
        pn[i] = term
        sum += pn[i]
        i += 1
    for i in n:
        pn[i] = pn[i] / sum
    return(pn)


class JobGen(Process):
    """ generates a sequence of Jobs
    """

    def execute(self, JobRate, MaxJob, mu):
        global NoInService, Busy
        for i in range(MaxJob):
            j = Job(sim=self.sim)
            self.sim.activate(j, j.execute(i, mu), delay=0.0)
            t = expovariate(JobRate)
            MT.tally(t)
            yield hold, self, t
        self.trace("Job generator finished")

    def trace(self, message):
        if JobGenTRACING:
            print("{0:8.4f} \t{1}".format(self.sim.now(), message))


class Job(Process):
    """ Jobs that are either accepted or rejected
    """

    def execute(self, i, mu):
        """ Job execution, only if accepted"""
        global NoInService, Busy, dist, NoRejected
```

```
        if NoInService < c:
            self.trace("Job %2d accepted b=%1d" % (i, Busy))
            NoInService += 1
            if NoInService == c:
                Busy = 1
                try:
                    BM.accum(Busy, self.sim.now())
                except:
                    "accum error BM=", BM
            # yield hold,self,Job.g.expovariate(self.mu);
            # dist= "Exponential"
            yield hold, self, ErlangVariate(1.0 / mu, 5)
            dist = "Erlang     "
            # yield hold,self,HyperVariate(1.0/8,m1=2.0,m2=4.0/7,g=Job.g);
            # dist= "HyperExpon "
            NoInService -= 1
            Busy = 0
            BM.accum(Busy, self.sim.now())
            self.trace("Job %2d leaving b=%1d" % (i, Busy))
        else:
            self.trace("Job %2d REJECT  b=%1d" % (i, Busy))
            NoRejected += 1

    def trace(self, message):
        if JobTRACING:
            print("{0:8.4f} \t{1}".format(self.sim.now(), message))


# Experiment data --------------------------
c = 2
lam = 1.0  # per minute
mu = 1.0 / 0.75  # per minute
p = 1.0 / 8
m1 = 2.0
m2 = 4.0 / 7.0
K = 5
rho = lam / mu

NoRejected = 0
NoInService = 0
Busy = 0

JobRate = lam
JobMax = 10000

JobTRACING = 0
JobGenTRACING = 0


# Model/Experiment ------------------------------

seed(111333)
s = Simulation()
BM = Monitor(sim=s)
MT = Monitor(sim=s)

s.initialize()
jbg = JobGen(sim=s)
```

```
s.activate(jbg, jbg.execute(1.0, JobMax, mu), 0.0)
s.simulate(until=20000.0)

# Analysis/output --------------------------

print('bcc')
print("time at the end = {0}".format(s.now()))
print("now = {0}\tstartTime = {1}".format(s.now(), BM.startTime))
print("No Rejected = {0:d}, ratio= {1}".format(
    NoRejected, (1.0 * NoRejected) / JobMax))
print("Busy proportion ({0}) = {1:8.6f}".format(dist, BM.timeAverage()))
print("Erlang pc (th)                = {0:8.6f}".format(erlangB(rho, c)[c]))
```

```
bcc
time at the end = 10085.751051963694
now = 10085.751051963694        startTime = 0.0
No Rejected = 1374, ratio= 0.1374
Busy proportion (Erlang     ) = 0.139016
Erlang pc (th)                = 0.138462
```

callCenter.py, callCenter_OO.py

Scenario: A call center runs around the clock. It has a number of agents online with different skills. Calls by clients with different questions arrive at an expected rate of callrate per minute (expo. distribution). An agent only deals with clients with questions in his competence areas. The number of agents online and their skills remain constant – when an agent goes offline, he is replaced by one with the same skills. The expected service time tService[i] per question follows an exponential distribution. Clients are impatient and renege if they don't get service within time tImpatience.

The model returns the frequency distribution of client waiting times, the percentage of reneging clients, and the load on the agents. This model demonstrates the use of yield get with a filter function. (KGM)

```
"""callCenter.py
Model shows use of get command with a filter function.

Scenario:
A call center runs around the clock. It has a number of agents online with
different skills/competences.
Calls by clients with different questions arrive at an expected rate of
callrate per minute (expo. distribution). An agent only deals with clients with
questions in his competence areas. The number of agents online and their skills
remain constant --
when an agent goes offline, he is replaced by one withe thesame skills.
The expected service time tService[i] per question
follows an exponential distribution.
Clients are impatient and renege if they don't get service within time
tImpatience.

* Determine the waiting times of clients.
* Determine the percentage renegers
* Determine the percentage load on agents.
"""
from SimPy.Simulation import *
import random as r
# Model components ------------------------------------------------------
```

```
class Client(Process):
    def __init__(self, need):
        Process.__init__(self)
        self.need = need

    def getServed(self, callCenter):
        self.done = SimEvent()
        callsWaiting = callCenter.calls
        self.served = False
        self.tArrive = now()
        yield put, self, callsWaiting, [self]
        yield hold, self, tImpatience
        # get here either after renege or after interrupt of
        # renege==successful call
        if self.interrupted():
            # success, got service
            callCenter.renegeMoni.observe(success)
            # wait for completion of service
            yield waitevent, self, self.done
        else:
            # renege
            callCenter.renegeMoni.observe(renege)
            callsWaiting.theBuffer.remove(self)
            callCenter.waitMoni.observe(now() - self.tArrive)
            if callsWaiting.monitored:
                callsWaiting.bufferMon.observe(y=len(callsWaiting.theBuffer))


class CallGenerator(Process):
    def __init__(self, name, center):
        Process.__init__(self, name)
        self.buffer = center.calls
        self.center = center

    def generate(self):
        while now() <= endTime:
            yield hold, self, r.expovariate(callrate)
            ran = r.random()
            for aNeed in clientNeeds:
                if ran < probNeeds[aNeed]:
                    need = aNeed
                    break
            c = Client(need=need)
            activate(c, c.getServed(callCenter=self.center))


class Agent(Process):
    def __init__(self, name, skills):
        Process.__init__(self, name)
        self.skills = skills
        self.busyMon = Monitor(name="Load on {0}".format(self.name))

    def work(self, callCtr):
        incoming = callCtr.calls

        def mySkills(buffer):
            ret = []
            for client in buffer:
```

```
                if client.need in self.skills:
                    ret.append(client)
                    break
            return ret
        self.started = now()
        while True:
            self.busyMon.observe(idle)
            yield get, self, incoming, mySkills
            self.busyMon.observe(busy)
            theClient = self.got[0]
            callCtr.waitMoni.observe(now() - theClient.tArrive)
            self.interrupt(theClient)  # interrupt the timeout renege
            yield hold, self, tService[theClient.need]
            theClient.done.signal()


class Callcenter:
    def __init__(self, name):
        self.calls = Store(name=name, unitName="call", monitored=True)
        self.waitMoni = Monitor("Caller waiting time")
        self.agents = []
        self.renegeMoni = Monitor("Renegers")


renege = 1
success = 0
busy = 1
idle = 0

# Experiment data -----------------------------------------------------------
centerName = "SimCityBank"
clientNeeds = ["loan", "insurance", "credit card", "other"]
aSkills = [["loan"], ["loan", "credit card"],
           ["insurance"], ["insurance", "other"]]
nrAgents = {0: 1, 1: 2, 2: 2, 3: 2}  # skill:nr agents of that skill
probNeeds = {"loan": 0.1, "insurance": 0.2, "credit card": 0.5, "other": 1.0}
tService = {"loan": 3., "insurance": 4.,
            "credit card": 2., "other": 3.}  # minutes
tImpatience = 3       # minutes
callrate = 7. / 10          # Callers per minute
endTime = 10 * 24 * 60    # minutes (10 days)
r.seed(12345)

# Model ---------------------------------------------------------------------


def model():
    initialize()
    callC = Callcenter(name=centerName)
    for i in nrAgents.keys():  # loop over skills
        for j in range(nrAgents[i]):  # loop over nr agents of that skill
            a = Agent(name="Agent type {0}".format(i), skills=aSkills[i])
            callC.agents.append(a)
            activate(a, a.work(callCtr=callC))
    # buffer=callC.calls)
    cg = CallGenerator(name="Call generator", center=callC)
    activate(cg, cg.generate())
    simulate(until=endTime)
```

```
    return callC


for tImpatience in (0.5, 1., 2.,):
    # Experiment --------------------------------------------------------------
    callCenter = model()
    # Analysis/output ---------------------------------------------------------
    print("\ntImpatience={0} minutes".format(tImpatience))
    print("==================")
    callCenter.waitMoni.setHistogram(low=0.0, high=float(tImpatience))
    try:
        print(callCenter.waitMoni.printHistogram(fmt="{0:6.1f}"))
    except:
        pass
    renegers = [1 for x in callCenter.renegeMoni.yseries() if x == renege]
    print("\nPercentage reneging callers: {0:4.1f}\n".format(
        100.0 * sum(renegers) / callCenter.renegeMoni.count()))
    for agent in callCenter.agents:
        print("Load on {0} (skills= {1}): {2:4.1f} percent".format(
            agent.name, agent.skills, agent.busyMon.timeAverage() * 100))
```

```
tImpatience=0.5 minutes
==================

Percentage reneging callers: 10.1

Load on Agent type 0 (skills= ['loan']): 13.9 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.1 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.1 percent
Load on Agent type 2 (skills= ['insurance']): 13.1 percent
Load on Agent type 2 (skills= ['insurance']): 13.1 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 44.3 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 44.3 percent

tImpatience=1.0 minutes
==================

Percentage reneging callers:  7.6

Load on Agent type 0 (skills= ['loan']): 13.5 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.4 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.4 percent
Load on Agent type 2 (skills= ['insurance']): 12.9 percent
Load on Agent type 2 (skills= ['insurance']): 12.9 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 46.0 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 45.8 percent

tImpatience=2.0 minutes
==================

Percentage reneging callers:  3.3

Load on Agent type 0 (skills= ['loan']): 13.8 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.8 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.8 percent
Load on Agent type 2 (skills= ['insurance']): 13.0 percent
Load on Agent type 2 (skills= ['insurance']): 13.0 percent
```

```
Load on Agent type 3 (skills= ['insurance', 'other']): 49.5 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 49.5 percent
```

Object Oriented version.

```
"""callCenter_OO.py
Model shows use of get command with a filter function.

Scenario:
A call center runs around the clock. It has a number of agents online with
different skills/competences.
Calls by clients with different questions arrive at an expected rate of
callrate per minute (expo. distribution). An agent only deals with clients with
questions in his competence areas. The number of agents online and their skills
remain
constant --
when an agent goes offline, he is replaced by one withe thesame skills.
The expected service time tService[i] per question
follows an exponential distribution.
Clients are impatient and renege if they don't get service within time
tImpatience.

* Determine the waiting times of clients.
* Determine the percentage renegers
* Determine the percentage load on agents.
"""
from SimPy.Simulation import *
import random as r
# Model components ------------------------------------------------------------


class Client(Process):
    def __init__(self, need, sim):
        Process.__init__(self, sim=sim)
        self.need = need

    def getServed(self, callCenter):
        self.done = SimEvent(sim=self.sim)
        callsWaiting = callCenter.calls
        self.served = False
        self.tArrive = self.sim.now()
        yield put, self, callsWaiting, [self]
        yield hold, self, tImpatience
        # get here either after renege or after interrupt of
        # renege==successful call
        if self.interrupted():
            # success, got service
            callCenter.renegeMoni.observe(success)
            # wait for completion of service
            yield waitevent, self, self.done
        else:
            # renege
            callCenter.renegeMoni.observe(renege)
            callsWaiting.theBuffer.remove(self)
            callCenter.waitMoni.observe(self.sim.now() - self.tArrive)
            if callsWaiting.monitored:
                callsWaiting.bufferMon.observe(y=len(callsWaiting.theBuffer))
```

```
class CallGenerator(Process):
    def __init__(self, name, center, sim):
        Process.__init__(self, name=name, sim=sim)
        self.buffer = center.calls
        self.center = center

    def generate(self):
        while self.sim.now() <= endTime:
            yield hold, self, r.expovariate(callrate)
            ran = r.random()
            for aNeed in clientNeeds:
                if ran < probNeeds[aNeed]:
                    need = aNeed
                    break
            c = Client(need=need, sim=self.sim)
            self.sim.activate(c, c.getServed(callCenter=self.center))


class Agent(Process):
    def __init__(self, name, skills, sim):
        Process.__init__(self, name=name, sim=sim)
        self.skills = skills
        self.busyMon = Monitor(
            name="Load on {0}".format(self.name), sim=self.sim)

    def work(self, callCtr):
        incoming = callCtr.calls

        def mySkills(buffer):
            ret = []
            for client in buffer:
                if client.need in self.skills:
                    ret.append(client)
                    break
            return ret
        self.started = self.sim.now()
        while True:
            self.busyMon.observe(idle)
            yield get, self, incoming, mySkills
            self.busyMon.observe(busy)
            theClient = self.got[0]
            callCtr.waitMoni.observe(self.sim.now() - theClient.tArrive)
            self.interrupt(theClient)  # interrupt the timeout renege
            yield hold, self, tService[theClient.need]
            theClient.done.signal()


class Callcenter:
    def __init__(self, name, sim):
        self.calls = Store(name=name, unitName="call", monitored=True, sim=sim)
        self.waitMoni = Monitor("Caller waiting time", sim=sim)
        self.agents = []
        self.renegeMoni = Monitor("Renegers", sim=sim)


renege = 1
success = 0
```

```
busy = 1
idle = 0

# Experiment data -----------------------------------------------------------
centerName = "SimCityBank"
clientNeeds = ["loan", "insurance", "credit card", "other"]
aSkills = [["loan"], ["loan", "credit card"],
           ["insurance"], ["insurance", "other"]]
nrAgents = {0: 1, 1: 2, 2: 2, 3: 2}  # skill:nr agents of that skill
probNeeds = {"loan": 0.1, "insurance": 0.2, "credit card": 0.5, "other": 1.0}
tService = {"loan": 3., "insurance": 4.,
            "credit card": 2., "other": 3.}  # minutes
tImpatienceRange = (0.5, 1., 2.,)      # minutes
callrate = 7. / 10          # Callers per minute
endTime = 10 * 24 * 60    # minutes (10 days)
r.seed(12345)

# Model ---------------------------------------------------------------------


class CallCenterModel(Simulation):
    def run(self):
        self.initialize()
        callC = Callcenter(name=centerName, sim=self)
        for i in nrAgents.keys():  # loop over skills
            for j in range(nrAgents[i]):  # loop over nr agents of that skill
                a = Agent(name="Agent type {0}".format(
                    i), skills=aSkills[i], sim=self)
                callC.agents.append(a)
                self.activate(a, a.work(callCtr=callC))
        # buffer=callC.calls)
        cg = CallGenerator(name="Call generator", center=callC, sim=self)
        self.activate(cg, cg.generate())
        self.simulate(until=endTime)
        return callC


for tImpatience in tImpatienceRange:
    # Experiment ----------------------------------------------------------
    callCenter = CallCenterModel().run()
    # Analysis/output -----------------------------------------------------
    print("\ntImpatience={0} minutes".format(tImpatience))
    print("==================")
    callCenter.waitMoni.setHistogram(low=0.0, high=float(tImpatience))
    try:
        print(callCenter.waitMoni.printHistogram(fmt="{0:6.1f}"))
    except:
        pass
    renegers = [1 for x in callCenter.renegeMoni.yseries() if x == renege]
    print("\nPercentage reneging callers: {0:4.1f}\n".format(
        100.0 * sum(renegers) / callCenter.renegeMoni.count()))
    for agent in callCenter.agents:
        print("Load on {0} (skills= {1}): {2:4.1f} percent".format(
            agent.name, agent.skills, agent.busyMon.timeAverage() * 100))
```

```
tImpatience=0.5 minutes
==================
```

```
Percentage reneging callers: 10.1

Load on Agent type 0 (skills= ['loan']): 13.9 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.1 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.1 percent
Load on Agent type 2 (skills= ['insurance']): 13.1 percent
Load on Agent type 2 (skills= ['insurance']): 13.1 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 44.3 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 44.3 percent

tImpatience=1.0 minutes
==================

Percentage reneging callers:  7.6

Load on Agent type 0 (skills= ['loan']): 13.5 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.4 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.4 percent
Load on Agent type 2 (skills= ['insurance']): 12.9 percent
Load on Agent type 2 (skills= ['insurance']): 12.9 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 46.0 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 45.8 percent

tImpatience=2.0 minutes
==================

Percentage reneging callers:  3.3

Load on Agent type 0 (skills= ['loan']): 13.8 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.8 percent
Load on Agent type 1 (skills= ['loan', 'credit card']): 24.8 percent
Load on Agent type 2 (skills= ['insurance']): 13.0 percent
Load on Agent type 2 (skills= ['insurance']): 13.0 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 49.5 percent
Load on Agent type 3 (skills= ['insurance', 'other']): 49.5 percent
```

cellphone.py, cellphone_OO.py

Simulate the operation of a BCC cellphone system. Calls arrive at random to a cellphone hub with a fixed number of channels. Service times are assumed exponential. The objective is to determine the statistics of busy periods in the operation of a BCC cellphone system.

The program simulates the operation for 10 observation periods and measures the mean and variance of the total time blocked, and the number of times blocking occurred in each hour. An observational gap occurs between the observation periods to make each one's measurement independent. (TV)

```
""" cellphone.py

Simulate the operation of a BCC cellphone system.

Calls arrive at random to a cellphone hub with a fixed number of
channels. Service times are assumed exponential. The objective
is to determine the statistics of busy periods in the operation of a
BCC cellphone system.
```

```
The required measurements are
(1) the total busy time (all channels full) in each 1-hour period and
(2) the total number of busy times in a 1-hour period.

The simulation is continuous but the observing Process, a
Statistician, breaks the time into 1-hour observation periods
separated by 15-minute gaps to reduce autocorrelation. The total busy
time and number of busy times in each interval is printed.

   """
from SimPy.Simulation import *
import random as ran


# Model components ------------------------

class CallSource(Process):
    """ generates a sequence of calls """

    def execute(self, maxN, lam, cell):
        for i in range(maxN):
            j = Call("Call{0:03d}".format(i))
            activate(j, j.execute(cell))
            yield hold, self, ran.expovariate(lam)


class Call(Process):
    """ Calls arrive at random at the cellphone hub"""

    def execute(self, cell):
        self.trace("arrived")
        if cell.Nfree == 0:
            self.trace("blocked and left")
        else:
            self.trace("got a channel")
            cell.Nfree -= 1
            if cell.Nfree == 0:
                self.trace("start busy period======")
                cell.busyStartTime = now()
                cell.totalBusyVisits += 1
            yield hold, self, ran.expovariate(mu)
            self.trace("finished")
            if cell.Nfree == 0:
                self.trace("end   busy period++++++")
                cell.busyEndTime = now()
                busy = now() - cell.busyStartTime
                self.trace("         busy  = {0:9.4f}".format(busy))
                cell.totalBusyTime += busy
            cell.Nfree += 1

    def trace(self, message):
        if TRACING:
            print("{0:7.4f} {1:13s} {2} ".format(now(), message, self.name))


class Cell:
    """ Holds global measurements"""
    Nfree = 0
```

```
    totalBusyTime = 0.0
    totalBusyVisits = 0
    result = ()


class Statistician(Process):
    """ observes the system at intervals """

    def execute(self, Nperiods, obsPeriod, obsGap, cell):
        cell.busyEndTime = now()  # simulation start time
        if STRACING:
            print("Busy time Number")
        for i in range(Nperiods):
            yield hold, self, obsGap
            cell.totalBusyTime = 0.0
            cell.totalBusyVisits = 0
            if cell.Nfree == 0:
                cell.busyStartTime = now()
            yield hold, self, obsPeriod
            if cell.Nfree == 0:
                cell.totalBusyTime += now() - cell.busyStartTime
            if STRACING:
                print("{0:7.3f} {1:5d}".format(
                    cell.totalBusyTime, cell.totalBusyVisits))
            m.tally(cell.totalBusyTime)
            bn.tally(cell.totalBusyVisits)
        stopSimulation()
        cell.result = (m.mean(), m.var(), bn.mean(), bn.var())

# Experiment data -------------------------


NChannels = 4          # number of channels in the cell
maxN = 10000
ranSeed = 3333333
lam = 1.0               # per minute
mu = 0.6667             # per minute
Nperiods = 10
obsPeriod = 60.0        # minutes
obsGap = 15.0        # gap between observation periods

TRACING = False
STRACING = True


# Experiment ------------------------------

m = Monitor()
bn = Monitor()
ran.seed(ranSeed)

cell = Cell()            # the cellphone tower
cell.Nfree = NChannels

initialize()
s = Statistician('Statistician')
activate(s, s.execute(Nperiods, obsPeriod, obsGap, cell))
g = CallSource('CallSource')
```

```
activate(g, g.execute(maxN, lam, cell))
simulate(until=10000.0)

# Output --------------------------
print('cellphone')
# input data:
print("lambda    mu      s  Nperiods obsPeriod  obsGap")
FMT = "{0:7.4f} {1:6.4f} {2:4d}   {3:4d}      {4:6.2f}   {5:6.2f}"
print(FMT.format(lam, mu, NChannels, Nperiods, obsPeriod, obsGap))


sr = cell.result
print("Busy Time:   mean = {0:6.3f} var= {1:6.3f}".format(sr[0], sr[1]))
print("Busy Number: mean = {0:6.3f} var= {1:6.3f}".format(sr[2], sr[3]))
```

```
Busy time Number
  5.857    15
  2.890     6
  1.219     4
  1.512     7
  4.237     5
  1.140     1
  3.596     8
  3.964     8
  4.146     9
  2.091     5
cellphone
lambda    mu      s  Nperiods obsPeriod  obsGap
 1.0000 0.6667    4     10       60.00    15.00
Busy Time:   mean =  3.065 var=  2.193
Busy Number: mean =  6.800 var= 12.360
```

Object orientated

```
""" cellphone_OO.py

Simulate the operation of a BCC cellphone system.

Calls arrive at random to a cellphone hub with a fixed number of
channels. Service times are assumed exponential. The objective
is to determine the statistics of busy periods in the operation of a
BCC cellphone system.

The required measurements are
(1) the total busy time (all channels full) in each 1-hour period and
(2) the total number of busy times in a 1-hour period.

The simulation is continuous but the observing Process, a
Statistician, breaks the time into 1-hour observation periods
separated by 15-minute gaps to reduce autocorrelation. The total busy
time and number of busy times in each interval is printed.

   """
from SimPy.Simulation import *
import random as ran


# Model components ------------------------
```

```
class CallSource(Process):
    """ generates a sequence of calls """

    def execute(self, maxN, lam, cell):
        for i in range(maxN):
            j = Call("Call{0:03d}".format(i), sim=self.sim)
            self.sim.activate(j, j.execute(cell))
            yield hold, self, ran.expovariate(lam)


class Call(Process):
    """ Calls arrive at random at the cellphone hub"""

    def execute(self, cell):
        self.trace("arrived")
        if cell.Nfree == 0:
            self.trace("blocked and left")
        else:
            self.trace("got a channel")
            cell.Nfree -= 1
            if cell.Nfree == 0:
                self.trace("start busy period======")
                cell.busyStartTime = self.sim.now()
                cell.totalBusyVisits += 1
            yield hold, self, ran.expovariate(mu)
            self.trace("finished")
            if cell.Nfree == 0:
                self.trace("end   busy period++++++")
                cell.busyEndTime = self.sim.now()
                busy = self.sim.now() - cell.busyStartTime
                self.trace("         busy  = {0:9.4f}".format(busy))
                cell.totalBusyTime += busy
            cell.Nfree += 1

    def trace(self, message):
        if TRACING:
            print("{0:7.4f} {1:13s} {2}".format(
                self.sim.now(), message, self.name))


class Cell:
    """ Holds global measurements"""
    Nfree = 0
    totalBusyTime = 0.0
    totalBusyVisits = 0
    result = ()


class Statistician(Process):
    """ observes the system at intervals """

    def execute(self, Nperiods, obsPeriod, obsGap, cell):
        cell.busyEndTime = self.sim.now()  # simulation start time
        if STRACING:
            print("Busy time Number")
        for i in range(Nperiods):
            yield hold, self, obsGap
```

```
            cell.totalBusyTime = 0.0
            cell.totalBusyVisits = 0
            if cell.Nfree == 0:
                cell.busyStartTime = self.sim.now()
            yield hold, self, obsPeriod
            if cell.Nfree == 0:
                cell.totalBusyTime += self.sim.now() - cell.busyStartTime
            if STRACING:
                print("{0:7.3f} {1:5d}".format(
                    cell.totalBusyTime, cell.totalBusyVisits))
            self.sim.m.tally(cell.totalBusyTime)
            self.sim.bn.tally(cell.totalBusyVisits)
        self.sim.stopSimulation()
        cell.result = (self.sim.m.mean(), self.sim.m.var(),
                       self.sim.bn.mean(), self.sim.bn.var())

# Experiment data -------------------------


NChannels = 4          # number of channels in the cell
maxN = 10000
ranSeed = 3333333
lam = 1.0               # per minute
mu = 0.6667             # per minute
Nperiods = 10
obsPeriod = 60.0        # minutes
obsGap = 15.0        # gap between observation periods

TRACING = False
STRACING = True

# Model -----------------------------------


class CellphoneModel(Simulation):
    def run(self):
        self.initialize()
        self.m = Monitor(sim=self)
        self.bn = Monitor(sim=self)
        ran.seed(ranSeed)
        self.cell = Cell()           # the cellphone tower
        self.cell.Nfree = NChannels
        s = Statistician('Statistician', sim=self)
        self.activate(s, s.execute(Nperiods, obsPeriod, obsGap, self.cell))
        g = CallSource('CallSource', sim=self)
        self.activate(g, g.execute(maxN, lam, self.cell))
        self.simulate(until=10000.0)


# Experiment ------------------------------
modl = CellphoneModel()
modl.run()

# Output ----------------------------------
print('cellphone')
# input data:
print("lambda    mu      s  Nperiods obsPeriod  obsGap")
FMT = "{0:7.4f} {1:6.4f} {2:4d}   {3:4d}      {4:6.2f}   {5:6.2f}"
```

```
print(FMT.format(lam, mu, NChannels, Nperiods, obsPeriod, obsGap))

sr = modl.cell.result
print("Busy Time:   mean = {0:6.3f} var= {1:6.3f}".format(sr[0], sr[1]))
print("Busy Number: mean = {0:6.3f} var= {1:6.3f}".format(sr[2], sr[3]))
```

```
Busy time Number
  5.857    15
  2.890     6
  1.219     4
  1.512     7
  4.237     5
  1.140     1
  3.596     8
  3.964     8
  4.146     9
  2.091     5
cellphone
lambda    mu      s  Nperiods obsPeriod  obsGap
 1.0000 0.6667    4     10       60.00    15.00
Busy Time:   mean =  3.065 var=  2.193
Busy Number: mean =  6.800 var= 12.360
```

Computer CPU: centralserver.py, centralserver_OO.py

A primitive central-server model with a single CPU and a single disk. A fixed number of users send "tasks" to the system which are processed and sent back to the user who then thinks for a time before sending a task back. Service times are exponential. This system can be solved analytically. (TV)

The user of each terminal thinks for a time (exponential, mean 100.0 sec) and then submits a task to the CPU with a service time (exponential, mean 1.0 sec). The user then remains idle until the task completes service and returns to him or her. The arriving tasks form a single FCFS queue in front of the CPU.

Upon leaving the CPU a task is either finished (probability 0.20) and returns to its user to begin another think time, or requires data from a disk drive (probability 0.8). If a task requires access to the disk, it joins a FCFS queue before service (service time at the disk, exponential, mean 1.39 sec). When finished with the disk, a task returns to the CPU queue again for another compute time (exp, mean 1.$ sec).

The objective is to measure the throughput of the CPU (tasks per second)

```
""" centralserver.py

A time-shared computer consists of a single
central processing unit (CPU) and a number of
terminals. The operator of each terminal `thinks'
for a time (exponential, mean 100.0 sec) and then
submits a task to the computer with a service time
(exponential, mean 1.0 sec). The operator then
remains idle until the task completes service and
returns to him or her. The arriving tasks form a
single FCFS queue in front of the CPU.

Upon leaving the CPU a task is either finished
(probability 0.20) and returns to its operator
to begin another `think' time, or requires data
from a disk drive (probability 0.8). If a task
```

```
requires access to the disk, it joins a FCFS queue
before service (service time at the disk,
exponential, mean 1.39 sec). When finished with
the disk, a task returns to the CPU queue again
for another compute time (exp, mean 1.$ sec).

the objective is to measure the throughput of
the CPU (tasks per second)
"""
from SimPy.Simulation import *
# from SimPy.SimulationTrace import *
import random as ran

# Model components ------------------------


class Task(Process):
    """ A computer  task  requires at least
    one use of the CPU and possibly accesses to a
    disk drive."""
    completed = 0
    rate = 0.0

    def execute(self, maxCompletions):
        while Task.completed < maxCompletions:
            self.debug(" starts thinking")
            thinktime = ran.expovariate(1.0 / MeanThinkTime)
            yield hold, self, thinktime
            self.debug(" request cpu")
            yield request, self, cpu
            self.debug(" got cpu")
            CPUtime = ran.expovariate(1.0 / MeanCPUTime)
            yield hold, self, CPUtime
            yield release, self, cpu
            self.debug(" finish cpu")
            while ran.random() < pDisk:
                self.debug(" request disk")
                yield request, self, disk
                self.debug(" got disk")
                disktime = ran.expovariate(1.0 / MeanDiskTime)
                yield hold, self, disktime
                self.debug(" finish disk")
                yield release, self, disk
                self.debug(" request cpu")
                yield request, self, cpu
                self.debug(" got cpu")
                CPUtime = ran.expovariate(1.0 / MeanCPUTime)
                yield hold, self, CPUtime
                yield release, self, cpu
            Task.completed += 1
        self.debug(" completed {0:d} tasks".format(Task.completed))
        Task.rate = Task.completed / float(now())

    def debug(self, message):
        FMT = "{0:9.3f} {1} {2}"
        if DEBUG:
            print(FMT.format(now(), self.name, message))
```

```
# Model ------------------------------
def main():
    initialize()
    for i in range(Nterminals):
        t = Task(name="task{0}".format(i))
        activate(t, t.execute(MaxCompletions))
    simulate(until=MaxrunTime)
    return (now(), Task.rate)

# Experiment data --------------------------


cpu = Resource(name='cpu')
disk = Resource(name='disk')
Nterminals = 3  # Number of terminals = Tasks
pDisk = 0.8  # prob. of going to disk
MeanThinkTime = 10.0  # seconds
MeanCPUTime = 1.0  # seconds
MeanDiskTime = 1.39  # seconds

ran.seed(111113333)
MaxrunTime = 20000.0
MaxCompletions = 100
DEBUG = False


# Experiment

result = main()

# Analysis/output --------------------------

print('centralserver')
print('{0:7.4f}: CPU rate = {1:7.4f} tasks per second'.format(
    result[0], result[1]))
```

```
centralserver
913.7400: CPU rate =  0.1116 tasks per second
```

OO version

```
""" centralserver.py

A time-shared computer consists of a single
central processing unit (CPU) and a number of
terminals. The operator of each terminal `thinks'
for a time (exponential, mean 100.0 sec) and then
submits a task to the computer with a service time
(exponential, mean 1.0 sec). The operator then
remains idle until the task completes service and
returns to him or her. The arriving tasks form a
single FCFS queue in front of the CPU.

Upon leaving the CPU a task is either finished
(probability 0.20) and returns to its operator
to begin another `think' time, or requires data
from a disk drive (probability 0.8). If a task
```

```
requires access to the disk, it joins a FCFS queue
before service (service time at the disk,
exponential, mean 1.39 sec). When finished with
the disk, a task returns to the CPU queue again
for another compute time (exp, mean 1.$ sec).

the objective is to measure the throughput of
the CPU (tasks per second)
"""
from SimPy.Simulation import *
# from SimPy.SimulationTrace import *
import random as ran

# Model components ------------------------


class Task(Process):
    """ A computer  task  requires at least
    one use of the CPU and possibly accesses to a
    disk drive."""
    completed = 0
    rate = 0.0

    def execute(self, maxCompletions):
        while Task.completed < MaxCompletions:
            self.debug(" starts thinking")
            thinktime = ran.expovariate(1.0 / MeanThinkTime)
            yield hold, self, thinktime
            self.debug(" request cpu")
            yield request, self, self.sim.cpu
            self.debug(" got cpu")
            CPUtime = ran.expovariate(1.0 / MeanCPUTime)
            yield hold, self, CPUtime
            yield release, self, self.sim.cpu
            self.debug(" finish cpu")
            while ran.random() < pDisk:
                self.debug(" request disk")
                yield request, self, self.sim.disk
                self.debug(" got disk")
                disktime = ran.expovariate(1.0 / MeanDiskTime)
                yield hold, self, disktime
                self.debug(" finish disk")
                yield release, self, self.sim.disk
                self.debug(" request cpu")
                yield request, self, self.sim.cpu
                self.debug(" got cpu")
                CPUtime = ran.expovariate(1.0 / MeanCPUTime)
                yield hold, self, CPUtime
                yield release, self, self.sim.cpu
            Task.completed += 1
        self.debug(" completed {0:d} tasks".format(Task.completed))
        Task.rate = Task.completed / float(self.sim.now())

    def debug(self, message):
        FMT = "({0:9.3f} {1} {2}"
        if DEBUG:
            print(FMT.format(self.sim.now(), self.name, message))
```

```
# Model ---------------------------
class CentralServerModel(Simulation):
    def run(self):
        self.initialize()
        self.cpu = Resource(name='cpu', sim=self)
        self.disk = Resource(name='disk', sim=self)
        for i in range(Nterminals):
            t = Task(name="task{0}".format(i), sim=self)
            self.activate(t, t.execute(MaxCompletions))
        self.simulate(until=MaxrunTime)
        return (self.now(), Task.rate)


# Experiment data -------------------------
Nterminals = 3  # Number of terminals = Tasks
pDisk = 0.8  # prob. of going to disk
MeanThinkTime = 10.0  # seconds
MeanCPUTime = 1.0  # seconds
MeanDiskTime = 1.39  # seconds

ran.seed(111113333)
MaxrunTime = 20000.0
MaxCompletions = 100
DEBUG = False


# Experiment

result = CentralServerModel().run()

# Analysis/output -------------------------

print('centralserver')
print('{0:7.4f}: CPU rate = {1:7.4f} tasks per second'.format(
    result[0], result[1]))
```

```
centralserver
913.7400: CPU rate =  0.1116 tasks per second
```

Messages on a Jackson Network: jacksonnetwork.py, jacksonnetwork_OO.py

A Jackson network with 3 nodes, exponential service times and probability switching. The simulation measures the delay for jobs moving through the system. (TV)

Messages arrive randomly at rate 1.5 per second at a communication network with 3 nodes (computers). Each computer (node) can queue messages.

```
"""
  jacksonnetwork.py

  Messages arrive randomly at rate 1.5 per second
  at a communication network with 3 nodes
  (computers). Each computer (node) can queue
  messages. Service-times are exponential with
  mean m_i at node i. These values are given in
```

```
  the column headed n_i in the table below.  On
  completing service at node i a message transfers
  to node j with probability p_ij and leaves the
  system with probability p_i3.

  These transition probabilities are as follows:
    Node     m_i  p_i0   p_i1  p_i2    p_i3
     i                                (leave)
     0       1.0   0      0.5  0.5     0
     1       2.0   0      0    0.8     0.2
     2       1.0   0.2    0    0       0.8

  Your task is to estimate
  (1) the average time taken for jobs going
      through the system and
  (2) the average number of jobs in the system.

"""
from SimPy.Simulation import *
# from SimPy.SimulationTrace import *
import random as ran

# Model components ------------------------


def choose2dA(i, P):
    """  return a random choice from a set j = 0..n-1
       with probs held in list of lists P[j] (n by n)
       using row i
       call:  next = choose2d(i,P)
    """
    U = ran.random()
    sumP = 0.0
    for j in range(len(P[i])):  # j = 0..n-1
        sumP += P[i][j]
        if U < sumP:
            break
    return(j)


class Msg(Process):
    """a message."""
    noInSystem = 0

    def execute(self, i):
        """ executing a message """
        startTime = now()
        Msg.noInSystem += 1
        # print("DEBUG noInSystm = ",Msg.noInSystem)
        NoInSystem.observe(Msg.noInSystem)
        self.trace("Arrived node {0}".format(i))
        while i != 3:
            yield request, self, node[i]
            self.trace("Got node {0}".format(i))
            st = ran.expovariate(1.0 / mean[i])
            yield hold, self, st
            yield release, self, node[i]
            self.trace("Finished with {0}".format(i))
```

```
            i = choose2dA(i, P)
            self.trace("Transfer to {0}".format(i))
        TimeInSystem.tally(now() - startTime)
        self.trace("leaving       {0} {1} in system".format(i, Msg.noInSystem))
        Msg.noInSystem -= 1
        NoInSystem.accum(Msg.noInSystem)

    def trace(self, message):
        if MTRACING:
            print("{0:7.4f} {1:3s} {2:10s}".format(now(), self.name, message))


class MsgSource(Process):
    """ generates a sequence of msgs """

    def execute(self, rate, maxN):
        self.count = 0   # hold number of messages generated
        self.trace("starting MsgSource")
        while (self.count < maxN):
            self.count += 1
            p = Msg("Message {0}".format(self.count))
            activate(p, p.execute(startNode))
            yield hold, self, ran.expovariate(rate)
        self.trace("generator finished with {0} ========".format(self.count))

    def trace(self, message):
        if GTRACING:
            print("{0:7.4f} \t{1}".format(now(), message))

# Experiment data -------------------------


rate = 1.5  # arrivals per second
maxNumber = 1000  # of Messages
GTRACING = False  # tracing Messages Source?

startNode = 0  # Messages always enter at node 0
ran.seed(77777)
MTRACING = False  # tracing Message action?

TimeInSystem = Monitor("time")
NoInSystem = Monitor("Number")

node = [Resource(1), Resource(1), Resource(1)]

mean = [1.0, 2.0, 1.0]  # service times, seconds
P = [[0, 0.5, 0.5, 0],  # transition matrix P_ij
     [0, 0, 0.8, 0.2],
     [0.2, 0, 0, 0.8]]

# Model/Experiment ------------------------------

initialize()
g = MsgSource(name="MsgSource")
activate(g, g.execute(rate, maxNumber))
simulate(until=5000.0)

# Analysis/output -------------------------
```

```
print('jacksonnetwork')
print("Mean number in system = {0:10.4f}".format(NoInSystem.timeAverage()))
print("Mean delay in system  = {0:10.4f}".format(TimeInSystem.mean()))
print("Total time run        = {0:10.4f}".format(now()))
print("Total jobs arrived    = {0:10d}".format(g.count))
print("Total jobs completed  = {0:10d}".format(TimeInSystem.count()))
print("Average arrival rate  = {0:10.4f}".format(g.count / now()))
```

```
jacksonnetwork
Mean number in system =   250.8179
Mean delay in system  =   310.7710
Total time run        =  1239.0304
Total jobs arrived    =       1000
Total jobs completed  =       1000
Average arrival rate  =     0.8071
```

OO version

```
"""
  jacksonnetwork_OO.py

  Messages arrive randomly at rate 1.5 per second
  at a communication network with 3 nodes
  (computers). Each computer (node) can queue
  messages. Service-times are exponential with
  mean m_i at node i. These values are given in
  the column headed n_i in the table below.  On
  completing service at node i a message transfers
  to node j with probability p_ij and leaves the
  system with probability p_i3.

  These transition probabilities are as follows:
    Node     m_i  p_i0   p_i1  p_i2    p_i3
     i                                (leave)
     0       1.0   0      0.5  0.5     0
     1       2.0   0      0    0.8     0.2
     2       1.0   0.2    0    0       0.8

  Your task is to estimate
  (1) the average time taken for jobs going
      through the system and
  (2) the average number of jobs in the system.

"""
from SimPy.Simulation import *
import random as ran

# Model components ------------------------


def choose2dA(i, P):
    """  return a random choice from a set j = 0..n-1
       with probs held in list of lists P[j] (n by n)
       using row i
       call:  next = choose2d(i,P)
    """
    U = ran.random()
```

```
    sumP = 0.0
    for j in range(len(P[i])):  # j = 0..n-1
        sumP += P[i][j]
        if U < sumP:
            break
    return(j)


class Msg(Process):
    """a message"""
    noInSystem = 0

    def execute(self, i):
        """ executing a message """
        startTime = self.sim.now()
        Msg.noInSystem += 1
        # print("DEBUG noInSystm = ",Msg.noInSystem)
        self.sim.NoInSystem.observe(Msg.noInSystem)
        self.trace("Arrived node  {0}".format(i))
        while i != 3:
            yield request, self, self.sim.node[i]
            self.trace("Got node {0}".format(i))
            st = ran.expovariate(1.0 / mean[i])
            yield hold, self, st
            yield release, self, self.sim.node[i]
            self.trace("Finished with {0}".format(i))
            i = choose2dA(i, P)
            self.trace("Transfer to   {0}".format(i))
        self.sim.TimeInSystem.tally(self.sim.now() - startTime)
        self.trace("leaving       {0} {1} in system".format(i, Msg.noInSystem))
        Msg.noInSystem -= 1
        self.sim.NoInSystem.accum(Msg.noInSystem)

    def trace(self, message):
        if MTRACING:
            print("{0:7.4f} {1:3d} {2:10s}".format(
                self.sim.now(), self.name, message))


class MsgSource(Process):
    """ generates a sequence of msgs """

    def execute(self, rate, maxN):
        self.count = 0   # hold number of messages generated
        while (self.count < maxN):
            self.count += 1
            p = Msg("Message {0}".format(self.count), sim=self.sim)
            self.sim.activate(p, p.execute(i=startNode))
            yield hold, self, ran.expovariate(rate)
        self.trace("generator finished with {0} ========".format(self.count))

    def trace(self, message):
        if GTRACING:
            print("{0:7.4f} \t{1}".format(self.sim.now(), message))

# Experiment data -------------------------
```

```
rate = 1.5  # arrivals per second
maxNumber = 1000  # of Messages
GTRACING = False  # tracing Messages Source?

startNode = 0  # Messages always enter at node 0
ran.seed(77777)
MTRACING = False  # tracing Message action?


mean = [1.0, 2.0, 1.0]  # service times, seconds
P = [[0, 0.5, 0.5, 0],  # transition matrix P_ij
     [0, 0, 0.8, 0.2],
     [0.2, 0, 0, 0.8]]

# Model ----------------------------------


class JacksonnetworkModel(Simulation):
    def run(self):
        self.initialize()
        self.TimeInSystem = Monitor("time", sim=self)
        self.NoInSystem = Monitor("Number", sim=self)
        self.node = [Resource(1, sim=self), Resource(
            1, sim=self), Resource(1, sim=self)]
        self.g = MsgSource("MsgSource", sim=self)
        self.activate(self.g, self.g.execute(rate, maxNumber))
        self.simulate(until=5000.0)


# Experiment -----------------------------
modl = JacksonnetworkModel()
modl.run()

# Analysis/output --------------------------

print('jacksonnetwork')
print("Mean number in system = {0:10.4f}".format(
    modl.NoInSystem.timeAverage()))
print("Mean delay in system  = {0:10.4f}".format(modl.TimeInSystem.mean()))
print("Total time run        = {0:10.4f}".format(modl.now()))
print("Total jobs arrived    = {0:10d}".format(modl.g.count))
print("Total jobs completed  = {0:10d}".format(modl.TimeInSystem.count()))
print("Average arrival rate  = {0:10.4f}".format(modl.g.count / modl.now()))
```

```
jacksonnetwork
Mean number in system =   250.8179
Mean delay in system  =   310.7710
Total time run        =  1239.0304
Total jobs arrived    =       1000
Total jobs completed  =       1000
Average arrival rate  =     0.8071
```

Miscellaneous Models

Bank Customers who can renege: bank08renege.py, bank08renege_OO.py

(Note currently does not run under Python 3)

Use of reneging (compound `yield request`) based on `bank08.py` of the tutorial TheBank. Customers leave if they lose patience with waiting.

```
""" bank08

 A counter with a random service time
 and customers who renege. Based on the program bank08.py
 from TheBank tutorial. (KGM)
"""
from SimPy.Simulation import *
from random import expovariate, seed, uniform

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval, counter):
        for i in range(number):
            c = Customer(name="Customer{0:02d}".format(i))
            activate(c, c.visit(counter, timeInBank=12.0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, counter, timeInBank=0):
        arrive = now()
        print("{0:7.4f} {1}: Here I am     ".format(now(), self.name))

        yield (request, self, counter), (hold, self, next(Customer.patience))

        if self.acquired(counter):
            wait = now() - arrive
            print("{0:7.4f} {1}: Waited {2:6.3f}".format(
                now(), self.name, wait))
            tib = expovariate(1.0 / timeInBank)
            yield hold, self, tib
            yield release, self, counter
            print("{0:7.4f} {1}: Finished".format(now(), self.name))
        else:
            wait = now() - arrive
            print("{0:7.4f} {1}: RENEGED after {2:6.3f}".format(
                now(), self.name, wait))

    def fpatience(minpatience=0, maxpatience=10000000000):
        while True:
            yield uniform(minpatience, maxpatience)
    fpatience = staticmethod(fpatience)
```

```
# Model --------------------------------------------


def model():
    counter = Resource(name="Karen")
    Customer.patience = Customer.fpatience(minpatience=1, maxpatience=3)
    initialize()
    source = Source('Source')
    activate(source, source.generate(NumCustomers,
                                     interval=IntervalCustomers,
                                     counter=counter))
    simulate(until=maxTime)

# Experiment data ------------------------


maxTime = 400.0
theseed = 1234
NumCustomers = 5
IntervalCustomers = 10.0
# Experiment --------------------------------

seed(theseed)
print('bank08renege')
model()
```

```
bank08renege
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 0.0902 Customer00: Finished
33.9482 Customer01: Here I am
33.9482 Customer01: Waited  0.000
44.4221 Customer01: Finished
58.1367 Customer02: Here I am
58.1367 Customer02: Waited  0.000
69.2708 Customer03: Here I am
70.3325 Customer03: RENEGED after  1.062
71.9733 Customer04: Here I am
73.6655 Customer04: RENEGED after  1.692
75.5906 Customer02: Finished
```

OO version

```
""" bank08_OO

 A counter with a random service time
 and customers who renege. Based on the program bank08.py
 from TheBank tutorial. (KGM)
"""
from SimPy.Simulation import *
from random import expovariate, seed, uniform

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""
```

```
    def generate(self, number, interval, counter):
        for i in range(number):
            c = Customer(name="Customer{0:02d}".format(i), sim=self.sim)
            self.sim.activate(c, c.visit(counter, timeInBank=12.0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, counter, timeInBank=0):
        arrive = self.sim.now()
        print("{0:7.4f} {1}: Here I am     ".format(self.sim.now(), self.name))

        yield (request, self, counter), (hold, self, next(Customer.patience))

        if self.acquired(counter):
            wait = self.sim.now() - arrive
            print("{0:7.4f} {1}: Waited {2:6.3f}".format(
                self.sim.now(), self.name, wait))
            tib = expovariate(1.0 / timeInBank)
            yield hold, self, tib
            yield release, self, counter
            print("{0:7.4f} {1}: Finished".format(self.sim.now(), self.name))
        else:
            wait = self.sim.now() - arrive
            print("{0:7.4f} {1}: RENEGED after {2:6.3f}".format(
                self.sim.now(), self.name, wait))

    def fpatience(minpatience=0, maxpatience=10000000000):
        while True:
            yield uniform(minpatience, maxpatience)
    fpatience = staticmethod(fpatience)

# Model --------------------------------------------


class BankModel(Simulation):
    def run(self):
        self.initialize()
        counter = Resource(name="Karen", sim=self)
        Customer.patience = Customer.fpatience(minpatience=1, maxpatience=3)
        source = Source(name='Source', sim=self)
        self.activate(source, source.generate(NumCustomers,
                                              interval=IntervalCustomers,
                                              counter=counter))
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 400.0
theseed = 1234
NumCustomers = 5
IntervalCustomers = 10.0
# Experiment ------------------------------
```

```
seed(theseed)
print('bank08renege')
BankModel().run()
```

```
bank08renege
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 0.0902 Customer00: Finished
33.9482 Customer01: Here I am
33.9482 Customer01: Waited  0.000
44.4221 Customer01: Finished
58.1367 Customer02: Here I am
58.1367 Customer02: Waited  0.000
69.2708 Customer03: Here I am
70.3325 Customer03: RENEGED after  1.062
71.9733 Customer04: Here I am
73.6655 Customer04: RENEGED after  1.692
75.5906 Customer02: Finished
```

Carwash: Carwash.py, Carwash_OO.py

Using a Store object for implementing master/slave cooperation between processes. Scenario is a carwash installation with multiple machines. Two model implementations are shown, one with the carwash as master in the cooperation, and the other with the car as master.

```
from SimPy.Simulation import *
import random
"""carwash.py
Scenario:
A carwash installation has nrMachines washing machines which wash a car in
washTime minutes.
Cars arrive with a negative exponential interarrival time with a mean of tInter
minutes.

Model the carwash operation as cooperation between two processes, the car being
washed and the machine doing the washing.

Build two implementations:
Model 1: the machine is master, the car slave;
Model 2: the car is master, the machine is slave.

"""
# Data:
nrMachines = 2
tInter = 2  # minutes
washtime = 3.5  # minutes
initialSeed = 123456
simTime = 100

####################################################
# Model 1: Carwash is master, car is slave
####################################################


class Carwash(Process):
```

```
    """Carwash machine; master"""

    def __init__(self, name):
        Process.__init__(self, name)
        self.carBeingWashed = None

    def lifecycle(self):
        while True:
            yield get, self, waitingCars, 1
            self.carBeingWashed = self.got[0]
            yield hold, self, washtime
            self.carBeingWashed.doneSignal.signal(self.name)


class Car(Process):
    """Car; slave"""

    def __init__(self, name):
        Process.__init__(self, name)
        self.doneSignal = SimEvent()

    def lifecycle(self):
        yield put, self, waitingCars, [self]
        yield waitevent, self, self.doneSignal
        whichWash = self.doneSignal.signalparam
        print("{0}: {1} done by {2}".format(now(), self.name, whichWash))


class CarGenerator(Process):
    """Car arrival generation"""

    def generate(self):
        i = 0
        while True:
            yield hold, self, r.expovariate(1.0 / tInter)
            c = Car("car{0}".format(i))
            activate(c, c.lifecycle())
            i += 1


print("Model 1: carwash is master")
print("--------------------------")
initialize()
r = random.Random()
r.seed(initialSeed)
waiting = []
for j in range(1, 5):
    c = Car("car-{0}".format(j))
    activate(c, c.lifecycle())
    waiting.append(c)
waitingCars = Store(capacity=40, initialBuffered=waiting)
cw = []
for i in range(2):
    c = Carwash("Carwash {0}".format(i))
    cw.append(c)
    activate(c, c.lifecycle())
cg = CarGenerator()
activate(cg, cg.generate())
```

```
simulate(until=simTime)
print("waiting cars: {0}".format([x.name for x in waitingCars.theBuffer]))
print("cars being washed: {0}".format([y.carBeingWashed.name for y in cw]))

##########################################################
# Model 2: Car is master, carwash is slave
##########################################################


class CarM(Process):
    """Car is master"""

    def __init__(self, name):
        Process.__init__(self, name)

    def lifecycle(self):
        yield get, self, washers, 1
        whichWash = self.got[0]
        carsBeingWashed.append(self)
        yield hold, self, washtime
        print("{0}: {1} done by {2}".format(now(), self.name, whichWash.name))
        whichWash.doneSignal.signal()
        carsBeingWashed.remove(self)


class CarwashS(Process):
    def __init__(self, name):
        Process.__init__(self, name)
        self.doneSignal = SimEvent()

    def lifecycle(self):
        while True:
            yield put, self, washers, [self]
            yield waitevent, self, self.doneSignal


class CarGenerator1(Process):
    def generate(self):
        i = 0
        while True:
            yield hold, self, r.expovariate(1.0 / tInter)
            c = CarM("car{0}".format(i))
            activate(c, c.lifecycle())
            i += 1


print("\nModel 2: car is master")
print("----------------------")
initialize()
r = random.Random()
r.seed(initialSeed)
washers = Store(capacity=nrMachines)
carsBeingWashed = []
for j in range(1, 5):
    c = CarM("car-{0}".format(j))
    activate(c, c.lifecycle())
for i in range(2):
    cw = CarwashS("Carwash {0}".format(i))
```

```
    activate(cw, cw.lifecycle())
cg = CarGenerator1()
activate(cg, cg.generate())
simulate(until=simTime)
print("waiting cars: {0}".format([x.name for x in washers.getQ]))
print("cars being washed: {0}".format([x.name for x in carsBeingWashed]))
```

```
Model 1: carwash is master
--------------------------
3.5: car-1 done by Carwash 0
3.5: car-2 done by Carwash 1
7.0: car-3 done by Carwash 0
7.0: car-4 done by Carwash 1
17.5: car0 done by Carwash 0
17.5: car1 done by Carwash 1
21.0: car2 done by Carwash 0
21.0: car3 done by Carwash 1
24.5: car4 done by Carwash 0
24.5: car5 done by Carwash 1
28.0: car6 done by Carwash 0
28.0: car7 done by Carwash 1
31.5: car8 done by Carwash 0
31.5: car9 done by Carwash 1
35.0: car10 done by Carwash 0
35.0: car11 done by Carwash 1
38.5: car12 done by Carwash 0
38.5: car13 done by Carwash 1
42.0: car14 done by Carwash 0
42.0: car15 done by Carwash 1
45.5: car16 done by Carwash 0
45.5: car17 done by Carwash 1
49.0: car18 done by Carwash 0
49.0: car19 done by Carwash 1
52.5: car20 done by Carwash 0
52.5: car21 done by Carwash 1
56.0: car22 done by Carwash 0
56.0: car23 done by Carwash 1
59.5: car24 done by Carwash 0
59.5: car25 done by Carwash 1
63.0: car26 done by Carwash 0
63.0: car27 done by Carwash 1
66.5: car28 done by Carwash 0
66.5: car29 done by Carwash 1
70.0: car30 done by Carwash 0
70.0: car31 done by Carwash 1
73.5: car32 done by Carwash 0
73.5: car33 done by Carwash 1
77.0: car34 done by Carwash 0
77.0: car35 done by Carwash 1
80.5: car36 done by Carwash 0
80.5: car37 done by Carwash 1
84.0: car38 done by Carwash 0
84.0: car39 done by Carwash 1
87.5: car40 done by Carwash 0
87.5: car41 done by Carwash 1
91.0: car42 done by Carwash 0
91.0: car43 done by Carwash 1
94.5: car44 done by Carwash 0
```

```
94.5: car45 done by Carwash 1
98.0: car46 done by Carwash 0
98.0: car47 done by Carwash 1
waiting cars: ['car50', 'car51', 'car52', 'car53', 'car54', 'car55', 'car56', 'car57',
↪ 'car58', 'car59']
cars being washed: ['car48', 'car49']

Model 2: car is master
----------------------
3.5: car-1 done by Carwash 0
3.5: car-2 done by Carwash 1
7.0: car-3 done by Carwash 0
7.0: car-4 done by Carwash 1
10.5: car0 done by Carwash 0
10.5: car1 done by Carwash 1
14.0: car2 done by Carwash 0
14.0: car3 done by Carwash 1
17.5: car4 done by Carwash 0
17.5: car5 done by Carwash 1
21.0: car6 done by Carwash 0
21.0: car7 done by Carwash 1
24.5: car8 done by Carwash 0
24.5: car9 done by Carwash 1
28.0: car10 done by Carwash 0
28.0: car11 done by Carwash 1
31.5: car12 done by Carwash 0
31.5: car13 done by Carwash 1
35.0: car14 done by Carwash 0
35.0: car15 done by Carwash 1
38.5: car16 done by Carwash 0
38.5: car17 done by Carwash 1
42.0: car18 done by Carwash 0
42.0: car19 done by Carwash 1
45.5: car20 done by Carwash 0
45.5: car21 done by Carwash 1
49.0: car22 done by Carwash 0
49.0: car23 done by Carwash 1
52.5: car24 done by Carwash 0
52.5: car25 done by Carwash 1
56.0: car26 done by Carwash 0
56.0: car27 done by Carwash 1
59.5: car28 done by Carwash 0
59.5: car29 done by Carwash 1
63.0: car30 done by Carwash 0
63.0: car31 done by Carwash 1
66.5: car32 done by Carwash 0
66.5: car33 done by Carwash 1
70.0: car34 done by Carwash 0
70.0: car35 done by Carwash 1
73.5: car36 done by Carwash 0
73.5: car37 done by Carwash 1
77.0: car38 done by Carwash 0
77.0: car39 done by Carwash 1
80.5: car40 done by Carwash 0
80.5: car41 done by Carwash 1
84.0: car42 done by Carwash 0
84.0: car43 done by Carwash 1
87.5: car44 done by Carwash 0
```

```
87.5: car45 done by Carwash 1
91.0: car46 done by Carwash 0
91.0: car47 done by Carwash 1
94.5: car48 done by Carwash 0
94.5: car49 done by Carwash 1
98.0: car50 done by Carwash 0
98.0: car51 done by Carwash 1
waiting cars: ['car54', 'car55', 'car56', 'car57', 'car58', 'car59']
cars being washed: ['car52', 'car53']
```

Here is the OO version:

```
from SimPy.Simulation import *
import random
"""Carwash_OO.py
Scenario:
A carwash installation has nrMachines washing machines which wash a car in
washTime minutes.
Cars arrive with a negative exponential interarrival time with a mean of tInter
minutes.

Model the carwash operation as cooperation between two processes, the car being
washed and the machine doing the washing.

Build two implementations:
Model 1: the machine is master, the car slave;
Model 2: the car is master, the machine is slave.

"""
# Experiment data --------------------------
nrMachines = 2
tInter = 2  # minutes
washtime = 3.5  # minutes
initialSeed = 123456
simTime = 100  # minutes

# Model 1 components ----------------------


class Carwash(Process):
    """Carwash machine; master"""

    def __init__(self, name, sim):
        Process.__init__(self, name=name, sim=sim)
        self.carBeingWashed = None

    def lifecycle(self):
        while True:
            yield get, self, self.sim.waitingCars, 1
            self.carBeingWashed = self.got[0]
            yield hold, self, washtime
            self.carBeingWashed.doneSignal.signal(self.name)


class Car(Process):
    """Car; slave"""

    def __init__(self, name, sim):
```

```
        Process.__init__(self, name=name, sim=sim)
        self.doneSignal = SimEvent(sim=sim)

    def lifecycle(self):
        yield put, self, self.sim.waitingCars, [self]
        yield waitevent, self, self.doneSignal
        whichWash = self.doneSignal.signalparam
        print("{0}: {1} done by {2}".format(
            self.sim.now(), self.name, whichWash))


class CarGenerator(Process):
    """Car arrival generation"""

    def generate(self):
        i = 0
        while True:
            yield hold, self, self.sim.r.expovariate(1.0 / tInter)
            c = Car("car{0}".format(i), sim=self.sim)
            self.sim.activate(c, c.lifecycle())
            i += 1

# Model 1: Carwash is master, car is slave


class CarWashModel1(Simulation):
    def run(self):
        print("Model 1: carwash is master")
        print("--------------------------")
        self.initialize()
        self.r = random.Random()
        self.r.seed(initialSeed)
        waiting = []
        for j in range(1, 5):
            c = Car("car{0}".format(-j), sim=self)
            self.activate(c, c.lifecycle())
            waiting.append(c)
        self.waitingCars = Store(
            capacity=40, initialBuffered=waiting, sim=self)
        cw = []
        for i in range(nrMachines):
            c = Carwash("Carwash {0}".format(i), sim=self)
            cw.append(c)
            self.activate(c, c.lifecycle())
        cg = CarGenerator(sim=self)
        self.activate(cg, cg.generate())
        self.simulate(until=simTime)

        print("waiting cars: {0}".format(
            [x.name for x in self.waitingCars.theBuffer]))
        print("cars being washed: {0}".format(
            [y.carBeingWashed.name for y in cw]))


# Experiment 1 ----------------------------
CarWashModel1().run()

##########################################
```

```
# Model 2 components ----------------------


class CarM(Process):
    """Car is master"""

    def lifecycle(self):
        yield get, self, self.sim.washers, 1
        whichWash = self.got[0]
        self.sim.carsBeingWashed.append(self)
        yield hold, self, washtime
        print("{0}: {1} done by {2}".format(
            self.sim.now(), self.name, whichWash.name))
        whichWash.doneSignal.signal()
        self.sim.carsBeingWashed.remove(self)


class CarwashS(Process):
    def __init__(self, name, sim):
        Process.__init__(self, name=name, sim=sim)
        self.doneSignal = SimEvent(sim=sim)

    def lifecycle(self):
        while True:
            yield put, self, self.sim.washers, [self]
            yield waitevent, self, self.doneSignal


class CarGenerator1(Process):
    def generate(self):
        i = 0
        while True:
            yield hold, self, self.sim.r.expovariate(1.0 / tInter)
            c = CarM("car{0}".format(i), sim=self.sim)
            self.sim.activate(c, c.lifecycle())
            i += 1

# Model 2: Car is master, carwash is slave


class CarWashModel2(Simulation):
    def run(self):
        print("\nModel 2: car is master")
        print("-----------------------")
        self.initialize()
        self.r = random.Random()
        self.r.seed(initialSeed)
        self.washers = Store(capacity=nrMachines, sim=self)
        self.carsBeingWashed = []
        for j in range(1, 5):
            c = CarM("car{0}".format(-j), sim=self)
            self.activate(c, c.lifecycle())
        for i in range(2):
            cw = CarwashS("Carwash {0}".format(i), sim=self)
            self.activate(cw, cw.lifecycle())
        cg = CarGenerator1(sim=self)
        self.activate(cg, cg.generate())
```

```
        self.simulate(until=simTime)

        print("waiting cars: {0}".format([x.name for x in self.washers.getQ]))
        print("cars being washed: {0}".format(
            [x.name for x in self.carsBeingWashed]))


# Experiment 1 ----------------------------
CarWashModel2().run()
```

```
Model 1: carwash is master
--------------------------
3.5: car-1 done by Carwash 0
3.5: car-2 done by Carwash 1
7.0: car-3 done by Carwash 0
7.0: car-4 done by Carwash 1
17.5: car0 done by Carwash 0
17.5: car1 done by Carwash 1
21.0: car2 done by Carwash 0
21.0: car3 done by Carwash 1
24.5: car4 done by Carwash 0
24.5: car5 done by Carwash 1
28.0: car6 done by Carwash 0
28.0: car7 done by Carwash 1
31.5: car8 done by Carwash 0
31.5: car9 done by Carwash 1
35.0: car10 done by Carwash 0
35.0: car11 done by Carwash 1
38.5: car12 done by Carwash 0
38.5: car13 done by Carwash 1
42.0: car14 done by Carwash 0
42.0: car15 done by Carwash 1
45.5: car16 done by Carwash 0
45.5: car17 done by Carwash 1
49.0: car18 done by Carwash 0
49.0: car19 done by Carwash 1
52.5: car20 done by Carwash 0
52.5: car21 done by Carwash 1
56.0: car22 done by Carwash 0
56.0: car23 done by Carwash 1
59.5: car24 done by Carwash 0
59.5: car25 done by Carwash 1
63.0: car26 done by Carwash 0
63.0: car27 done by Carwash 1
66.5: car28 done by Carwash 0
66.5: car29 done by Carwash 1
70.0: car30 done by Carwash 0
70.0: car31 done by Carwash 1
73.5: car32 done by Carwash 0
73.5: car33 done by Carwash 1
77.0: car34 done by Carwash 0
77.0: car35 done by Carwash 1
80.5: car36 done by Carwash 0
80.5: car37 done by Carwash 1
84.0: car38 done by Carwash 0
84.0: car39 done by Carwash 1
87.5: car40 done by Carwash 0
87.5: car41 done by Carwash 1
```

```
91.0: car42 done by Carwash 0
91.0: car43 done by Carwash 1
94.5: car44 done by Carwash 0
94.5: car45 done by Carwash 1
98.0: car46 done by Carwash 0
98.0: car47 done by Carwash 1
waiting cars: ['car50', 'car51', 'car52', 'car53', 'car54', 'car55', 'car56', 'car57',
↪ 'car58', 'car59']
cars being washed: ['car48', 'car49']

Model 2: car is master
----------------------
3.5: car-1 done by Carwash 0
3.5: car-2 done by Carwash 1
7.0: car-3 done by Carwash 0
7.0: car-4 done by Carwash 1
10.5: car0 done by Carwash 0
10.5: car1 done by Carwash 1
14.0: car2 done by Carwash 0
14.0: car3 done by Carwash 1
17.5: car4 done by Carwash 0
17.5: car5 done by Carwash 1
21.0: car6 done by Carwash 0
21.0: car7 done by Carwash 1
24.5: car8 done by Carwash 0
24.5: car9 done by Carwash 1
28.0: car10 done by Carwash 0
28.0: car11 done by Carwash 1
31.5: car12 done by Carwash 0
31.5: car13 done by Carwash 1
35.0: car14 done by Carwash 0
35.0: car15 done by Carwash 1
38.5: car16 done by Carwash 0
38.5: car17 done by Carwash 1
42.0: car18 done by Carwash 0
42.0: car19 done by Carwash 1
45.5: car20 done by Carwash 0
45.5: car21 done by Carwash 1
49.0: car22 done by Carwash 0
49.0: car23 done by Carwash 1
52.5: car24 done by Carwash 0
52.5: car25 done by Carwash 1
56.0: car26 done by Carwash 0
56.0: car27 done by Carwash 1
59.5: car28 done by Carwash 0
59.5: car29 done by Carwash 1
63.0: car30 done by Carwash 0
63.0: car31 done by Carwash 1
66.5: car32 done by Carwash 0
66.5: car33 done by Carwash 1
70.0: car34 done by Carwash 0
70.0: car35 done by Carwash 1
73.5: car36 done by Carwash 0
73.5: car37 done by Carwash 1
77.0: car38 done by Carwash 0
77.0: car39 done by Carwash 1
80.5: car40 done by Carwash 0
80.5: car41 done by Carwash 1
```

```
84.0: car42 done by Carwash 0
84.0: car43 done by Carwash 1
87.5: car44 done by Carwash 0
87.5: car45 done by Carwash 1
91.0: car46 done by Carwash 0
91.0: car47 done by Carwash 1
94.5: car48 done by Carwash 0
94.5: car49 done by Carwash 1
98.0: car50 done by Carwash 0
98.0: car51 done by Carwash 1
waiting cars: ['car54', 'car55', 'car56', 'car57', 'car58', 'car59']
cars being washed: ['car52', 'car53']
```

Game of Life: CellularAutomata.py

A two-dimensional cellular automaton. Does the game of Life. (KGM)

```
from SimPy.Simulation import *
"""CellularAutomata.py
Simulation of two-dimensional cellular automata. Plays game of Life.
"""


class Autom(Process):
    def __init__(self, coords):
        Process.__init__(self)
        self.x = coords[0]
        self.y = coords[1]
        self.state = False

    def nrActiveNeighbours(self, x, y):
        nr = 0
        coords = [(xco + x, yco + y) for xco in (-1, 0, 1)
                  for yco in (-1, 0, 1) if not (xco == 0 and yco == 0)]

        for a_coord in coords:
            try:
                if cells[a_coord].state:
                    nr += 1
            except KeyError:
                # wrap around
                nux = divmod(a_coord[0], size)[1]
                nuy = divmod(a_coord[1], size)[1]
                if cells[(nux, nuy)].state:
                    nr += 1
        return nr

    def decide(self, nrActive):
        return (self.state and (nrActive == 2 or nrActive == 3) or
                               (nrActive == 3))

    def celllife(self):
        while True:
            # calculate next state
            temp = self.decide(self.nrActiveNeighbours(self.x, self.y))
            yield hold, self, 0.5
```

```
            # set next state
            self.state = temp
            yield hold, self, 0.5


class Show(Process):
    def __init__(self):
        Process.__init__(self)

    def picture(self):
        while True:
            print("Generation {0}".format(now()))
            for i in range(size):
                cls = " "
                for j in range(size):
                    if cells[(i, j)].state:
                        cls += " *"
                    else:
                        cls += " ."
                print(cls)
            print("")
            yield hold, self, 1


size = 20
cells = {}
initialize()
for i in range(size):
    for j in range(size):
        a = cells[(i, j)] = Autom((i, j))
        activate(a, a.celllife())

# R-pentomino
cells[(9, 3)].state = True
cells[(10, 3)].state = True
cells[(9, 4)].state = True
cells[(8, 4)].state = True
cells[(9, 5)].state = True

cells[(5, 5)].state = True
cells[(5, 6)].state = True
cells[(4, 5)].state = True
cells[(4, 6)].state = True
cells[(4, 7)].state = True
cells[(10, 10)].state = True
cells[(10, 11)].state = True
cells[(10, 12)].state = True
cells[(10, 13)].state = True
cells[(11, 10)].state = True
cells[(11, 11)].state = True
cells[(11, 12)].state = True
cells[(11, 13)].state = True

print('CellularAutomata')
s = Show()
whenToStartShowing = 10
activate(s, s.picture(), delay=whenToStartShowing)
nrGenerations = 30
```

```
simulate(until=nrGenerations)
```

```
CellularAutomata
Generation 10
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . . .
  . . * . * * . . . . . . . . . . . . . . .
  . . . . . * . . . . . . . . . . . . . . .
  . . . . . * * * . . . * * . . . . . . . .
  . . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .

Generation 11
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . . .
  . . * . . * . . . . . . . . . . . . . . .
  . . * . . * . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . * * . . . . * * . . . . . . . .
  . . . . . . * . . . * . . * . . . . . . .
  . . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .

Generation 12
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . . .
  . * * . . * . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . * * . . . . . . . . . . . . . .
  . . . . . * * . . . . * * . . . . . . . .
```

```
  . . . . . * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 13
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  . * * . * . . . . . . . . . . . . . . .
  . . . . . * * . . . . . . . . . . . . .
  . . . . . * * . . . . . . . . . . . . .
  . . . . * . . * . . . * * . . . . . . .
  . . . . . * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 14
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * * . . . . . . . . . . . . . .
  . * * * * . . . . . . . . . . . . . . .
  . . . . * . * . . . . . . . . . . . . .
  . . . . * . . * . . . . . . . . . . . .
  . . . . * . . * . . . * * . . . . . . .
  . . . . . * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 15
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
```

```
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . . * . . * . . . . . . . . . . . . . .
  . * . . . * . . . . . . . . . . . . . .
  . * * . . . . . . . . . . . . . . . . .
  . . * . * . . . . . . . . . . . . . . .
  . . . * * . * * . . . . . . . . . . . .
  . . . . * . . * . . . * * . . . . . . .
  . . . . . * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 16
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * * . . * . . . . . . . . . . . . . .
  . * . . . . . . . . . . . . . . . . . .
  . * * * . . . . . . . . . . . . . . . .
  . * * . * * . . . . . . . . . . . . . .
  . . . . * . * * . . . . . . . . . . . .
  . . . * * . . * . . . * * . . . . . . .
  . . . . . * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 17
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  * . . * . . . . . . . . . . . . . . . .
  * . . * * . . . . . . . . . . . . . . .
  . * . . * * . . . . . . . . . . . . . .
  . . * . . * * . . . . . . . . . . . . .
  . . . * * . . * . . . * * . . . . . . .
  . . . . * * * . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * * . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
```

```
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .

Generation 18
  . . . . . .
  . . . * . .
  . . * * * .
  . * . . * *
  * * * * * .
  * * * * . .
  * * * * . .
  . * * . * .
  . . * . . .
  . . . * * .
  . . . * * *
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .

Generation 19
  . . . . . .
  . . * * * .
  . . * . . *
  * . . . . *
  . . . . . *
  . . . . . .
  . . . . * .
  * . . . . *
  . * * . * *
  . . * . . *
  . . . * . *
  . . . . * *
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .
  . . . . . .

Generation 20
  . . . * . .
  . . * * * .
  . * * . . *
  . . . . * *
  . . . . . .
  . . . . . .
  . . . . . .
  . * . * * .
```

```
  . * * * . * . . * . . . . . . . . . . .
  . * * . * * . * * . . * * . . . . . . .
  . . . . . * * * . . * . . * . . . . . .
  . . . . . * * . . . . * . . * . . . . .
  . . . . . . . . . . . . * * . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 21
  . . * * * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  . * * . . . * . . . . . . . . . . . . .
  . . . . * * * . . . . . . . . . . . . .
  . . . . . * . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . * . . . . . . . . . . . .
  . * . * * * * * . . . . . . . . . . . .
  * . . . . . . . * . . . . . . . . . . .
  . * . . . . . . * * . * * . . . . . . .
  . . . . . . . . * * * . . * . . . . . .
  . . . . . * . * . . * . . * . . . . . .
  . . . . . . . . . . * * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 22
  . . * * * . . . . . . . . . . . . . . .
  . * . . * * . . . . . . . . . . . . . .
  . * * * * . * . . . . . . . . . . . . .
  . . . . * . * . . . . . . . . . . . . .
  . . . . * * * . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . * * . * * . . . . . . . . . . .
  . . . . * * * * * . . . . . . . . . . .
  * * * . * * * . . * * . . . . . . . . .
  . . . . . . . . . . * * . . . . . . . .
  . . . . . . . * . . . . * . . . . . . .
  . . . . . . . . * . * . . * . . . . . .
  . . . . . . . . . . * * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .

Generation 23
```

```
  . . * . . * . . . . . . . . . . . . . . . . . .
  . * . . . . . . . . . . . . . . . . . . . . . .
  . * * . . . * . . . . . . . . . . . . . . . . .
  . . * . . . * * . . . . . . . . . . . . . . . .
  . . . . * . * . . . . . . . . . . . . . . . . .
  . . . . . . . * . . . . . . . . . . . . . . . .
  . . . . * . . . * . . . . . . . . . . . . . . .
  . * . . . . . . . . . . . . . . . . . . . . . .
  . * . * * . . . * * * * . . . . . . . . . . . .
  . * . . . * * . . . * * * . . . . . . . . . . .
  . . . . . . . . . . . * . * . . . . . . . . . .
  . . . . . . . . . . . * . * . . . . . . . . . .
  . . . . . . . . . . . * * . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . . . . . .

Generation 24
  . * * . * . . . . . . . . . . . . . . . . . . .
  . * . . . . . . . . . . . . . . . . . . . . . .
  . * * . . . * * . . . . . . . . . . . . . . . .
  . * * * . . * * . . . . . . . . . . . . . . . .
  . . . . . * * . . . . . . . . . . . . . . . . .
  . . . . . * . * . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . * . * . . . . . . . . . . . . .
  * * . . * * . . . * . . * . . . . . . . . . . .
  . . * . * * . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . * . . . . . . . . . .
  . . . . . . . . . . * * . * . . . . . . . . . .
  . . . . . . . . . . * * . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . . . . . .

Generation 25
  . * . . * . . . . . . . . . . . . . . . . . . .
  * . . * . . . . . . . . . . . . . . . . . . . .
  * . . * . . * * . . . . . . . . . . . . . . . .
  . * . * . . . . . . . . . . . . . . . . . . . .
  . . * . * * . . . . . . . . . . . . . . . . . .
  . . . . . * . . . . . . . . . . . . . . . . . .
  . . . * * . . . . . . . . . . . . . . . . . . .
  . * * * * * . . . * . . . . . . . . . . . . . .
  . * . . . . . . . * . . . . . . . . . . . . . .
  . * . * * * . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . * . . . . . . . . . . . .
  . . . . . . . . . . * * . * . . . . . . . . . .
  . . . . . . . . . . * * * . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . .
```

```
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .

Generation 26
  * * * * * . . . . . . . . . . . . . . .
  * * * * * . . . . . . . . . . . . . . .
  * * . * * . . . . . . . . . . . . . . .
  . * . * . * * . . . . . . . . . . . . .
  . . * * * * . . . . . . . . . . . . . .
  . . . . . * . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . * . . . * . . . . . . . . . . . . . .
  * * . . . . . . . . . . . . . . . . . .
  . . * . * . . . . . . . . . . . . . . .
  . . . . * . . . . . * * . . . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . * . * . . . . . . .
  . . . . . . . . . . . * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * * . . . . . . . . . . . . . .

Generation 27
  . . . . . . . . . . . . . . . . . . . .
  . . . . . * . . . . . . . . . . . . . *
  . . . . . . . . . . . . . . . . . . . .
  * * . . . . * . . . . . . . . . . . . .
  . . * * . . . . . . . . . . . . . . . .
  . . . * . * . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  * * . . . . . . . . . . . . . . . . . .
  * * * . . . . . . . . . . . . . . . . .
  . * . * . . . . . . . . . . . . . . . .
  . . . * . . . . . . . * * . . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . * . * . . . . . . .
  . . . . . . . . . . . * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . . * . . * . . . . . . . . . . . . . .
  * . . . . * . . . . . . . . . . . . . .

Generation 28
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  * . . . . . . . . . . . . . . . . . . .
  . * * . . . . . . . . . . . . . . . . .
  . * * * * . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
```

```
  . . . . . . . . . . . . . . . . . . . .
  * . * . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  * * . * . . . . . . . . . . . . . . . .
  . . * . . . . . . . . * * . . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * * . . * . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 29
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . * . . . . . . . . . . . . . . . . . .
  * . . . . . . . . . . . . . . . . . . .
  . . . . * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  . * * . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  * . * . . . . . . . . . . . . . . . . .
  . * * . . . . . . . . . . . . . . . . .
  . * * . . . . . . . . * * . . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * . . . . . . . . . . . . . . .
  . * * . * . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .

Generation 30
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . * * * . . . . . . . . . . . . . . . .
  . * * . . . . . . . . . . . . . . . . .
  . . * . . . . . . . . . . . . . . . . .
  . . * . . . . . . . . . . . . . . . . .
  * . . * . . . . . . . . . . . . . . . .
  . * * . . . . . . . . * * . . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . * . . * . . . . . .
  . . . . . . . . . . . * . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
  . . . * . . . . . . . . . . . . . . . .
  . . * * * . . . . . . . . . . . . . . .
  . * . . * * . . . . . . . . . . . . . .
  . * * * . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . .
```

SimPy's event signalling synchronisation constructs: demoSimPyEvents.py

Demo of the event signalling constructs. Three small simulations are included: Pavlov's drooling dogs, an activity simulation where a job is completed after a number of parallel activities, and the simulation of a US-style 4-way stop intersection.

```
from SimPy.Simulation import *
import random
"""
   Demo of SimPy's event signalling synchronization constructs
"""

print('demoSimPyEvents')

# Pavlov's dogs
"""Scenario:
Dogs start to drool when Pavlov rings the bell.
"""


class BellMan(Process):
    def ring(self):
        while True:
            bell.signal()
            print("{0} {1} rings bell".format(now(), self.name))
            yield hold, self, 5


class PavlovDog(Process):
    def behave(self):
        while True:
            yield waitevent, self, bell
            print("{0} {1} drools".format(now(), self.name))


random.seed(111333555)
initialize()
bell = SimEvent("bell")
for i in range(4):
    p = PavlovDog("Dog {0}".format(i + 1))
    activate(p, p.behave())
b = BellMan("Pavlov")
activate(b, b.ring())
print("\n Pavlov's dogs")
simulate(until=10)

# PERT simulation
"""
Scenario:
A job (TotalJob) requires 10 parallel activities with random duration to be
completed.
"""
```

```
class Activity(Process):
    def __init__(self, name):
        Process.__init__(self, name)
        self.event = SimEvent("completion of {0}".format(self.name))
        allEvents.append(self.event)

    def perform(self):
        yield hold, self, random.randint(1, 100)
        self.event.signal()
        print("{0} Event '{1}' fired".format(now(), self.event.name))


class TotalJob(Process):
    def perform(self, allEvents):
        for e in allEvents:
            yield waitevent, self, e
        print(now(), "All done")


random.seed(111333555)
initialize()
allEvents = []
for i in range(10):
    a = Activity("Activity {0}".format(i + 1))
    activate(a, a.perform())
t = TotalJob()
activate(t, t.perform(allEvents))
print("\n PERT network simulation")
simulate(until=100)

# US-style4-way stop intersection
"""
Scenario:
At a US-style 4-way stop intersection, a car may only enter the intersection
when it is free.
Cars enter in FIFO manner.
"""


class Car(Process):
    def drive(self):
        print("{0:4.1f} {1} waiting to enter intersection".format(now(),
                                                                  self.name))
        yield queueevent, self, intersectionFree
        # Intersection free, enter  . .
        # Begin Critical Section
        yield hold, self, 1  # drive across
        print("{0:4.1f} {1} crossed intersection".format(now(), self.name))
        # End Critical Section
        intersectionFree.signal()


random.seed(111333555)
initialize()
intersectionFree = SimEvent("Intersection free")
intersectionFree.signal()
arrtime = 0.0
for i in range(20):
```

```
    c = Car("Car {0}".format(i + 1))
    activate(c, c.drive(), at=arrtime)
    arrtime += 0.2
print("\n 4-way stop intersection")
print(simulate(until=100))
```

```
demoSimPyEvents

 Pavlov's dogs
0 Pavlov rings bell
0 Dog 4 drools
0 Dog 3 drools
0 Dog 2 drools
0 Dog 1 drools
5 Pavlov rings bell
5 Dog 1 drools
5 Dog 2 drools
5 Dog 3 drools
5 Dog 4 drools
10 Pavlov rings bell
10 Dog 4 drools
10 Dog 3 drools
10 Dog 2 drools
10 Dog 1 drools

 PERT network simulation
13 Event 'completion of Activity 6' fired
19 Event 'completion of Activity 10' fired
22 Event 'completion of Activity 7' fired
42 Event 'completion of Activity 1' fired
48 Event 'completion of Activity 4' fired
55 Event 'completion of Activity 8' fired
69 Event 'completion of Activity 5' fired
83 Event 'completion of Activity 2' fired
90 Event 'completion of Activity 9' fired
93 Event 'completion of Activity 3' fired
93 All done

 4-way stop intersection
 0.0 Car 1 waiting to enter intersection
 0.2 Car 2 waiting to enter intersection
 0.4 Car 3 waiting to enter intersection
 0.6 Car 4 waiting to enter intersection
 0.8 Car 5 waiting to enter intersection
 1.0 Car 6 waiting to enter intersection
 1.0 Car 1 crossed intersection
 1.2 Car 7 waiting to enter intersection
 1.4 Car 8 waiting to enter intersection
 1.6 Car 9 waiting to enter intersection
 1.8 Car 10 waiting to enter intersection
 2.0 Car 11 waiting to enter intersection
 2.0 Car 2 crossed intersection
 2.2 Car 12 waiting to enter intersection
 2.4 Car 13 waiting to enter intersection
 2.6 Car 14 waiting to enter intersection
 2.8 Car 15 waiting to enter intersection
 3.0 Car 3 crossed intersection
 3.0 Car 16 waiting to enter intersection
```

```
 3.2 Car 17 waiting to enter intersection
 3.4 Car 18 waiting to enter intersection
 3.6 Car 19 waiting to enter intersection
 3.8 Car 20 waiting to enter intersection
 4.0 Car 4 crossed intersection
 5.0 Car 5 crossed intersection
 6.0 Car 6 crossed intersection
 7.0 Car 7 crossed intersection
 8.0 Car 8 crossed intersection
 9.0 Car 9 crossed intersection
10.0 Car 10 crossed intersection
11.0 Car 11 crossed intersection
12.0 Car 12 crossed intersection
13.0 Car 13 crossed intersection
14.0 Car 14 crossed intersection
15.0 Car 15 crossed intersection
16.0 Car 16 crossed intersection
17.0 Car 17 crossed intersection
18.0 Car 18 crossed intersection
19.0 Car 19 crossed intersection
20.0 Car 20 crossed intersection
SimPy: No more events at time 20
```

Find the Shortest Path: shortestPath_SimPy.py, shortestPath_SimPy_OO.py

A fun example of using SimPy for non-queuing work. It simulates a searcher through a graph, seeking the shortest path. (KGM)

```
from SimPy.Simulation import *
""" shortestPath_SimPy.py

    Finds the shortest path in a network.
    Author: Klaus Muller
"""


class node:
    def __init__(self):
        self.reached = 0


class searcher(Process):
    def __init__(self, graph, path, length, from_node, to_node,
                 distance, goal_node):
        Process.__init__(self)
        self.path = path[:]
        self.length = length
        self.from_node = from_node
        self.to_node = to_node
        self.distance = distance
        self.graph = graph
        self.goal_node = goal_node

    def run(self, to_node):
        if DEMO:
            print("Path so far: {0} (length {1}). "
```

```
                    "Search from {2} to {3}".format(self.path, self.length,
                                                    self.from_node, to_node))
            yield hold, self, self.distance
            if not nodes[to_node].reached:
                self.path.append(to_node)
                self.length += self.distance
                nodes[to_node].reached = 1
                if to_node == self.goal_node:
                    print("SHORTEST PATH", self.path, "Length:", self.length)
                    stopSimulation()
                else:
                    for i in self.graph[to_node]:
                        s = searcher(graph=self.graph, path=self.path,
                                     length=self.length, from_node=i[0],
                                     to_node=i[1], distance=i[2],
                                     goal_node=self.goal_node)
                        activate(s, s.run(i[1]))


print('shortestPath_SimPy')
initialize()
nodes = {}
DEMO = 1
for i in ("Atown", "Btown", "Ccity", "Dpueblo", "Evillage", "Fstadt"):
    nodes[i] = node()
""" Format graph definition:
a_graph={node_id:[(from,to,distance),
                  (from,to,distance)],
                  node_id:[ . . . ])
"""
net = {"Atown": (("Atown", "Btown", 3.5), ("Atown", "Ccity", 1),
                 ("Atown", "Atown", 9), ("Atown", "Evillage", 0.5)),
       "Btown": (("Btown", "Ccity", 5),),
       "Ccity": (("Ccity", "Ccity", 1), ("Ccity", "Fstadt", 9),
                 ("Ccity", "Dpueblo", 3), ("Ccity", "Atown", 3)),
       "Dpueblo": (("Dpueblo", "Ccity", 2), ("Dpueblo", "Fstadt", 10)),
       "Evillage": (("Evillage", "Btown", 1),),
       "Fstadt": (("Fstadt", "Ccity", 3),)}
if DEMO:
    print("Search for shortest path from {0} to {1} \nin graph {2}".format(
        "Atown", "Fstadt", sorted(net.items())))
startup = searcher(graph=net, path=[], length=0, from_node="Atown",
                   to_node="Atown", distance=0, goal_node="Fstadt")
activate(startup, startup.run("Atown"))
simulate(until=10000)
```

```
shortestPath_SimPy
Search for shortest path from Atown to Fstadt
in graph [('Atown', (('Atown', 'Btown', 3.5), ('Atown', 'Ccity', 1), ('Atown', 'Atown
↪', 9), ('Atown', 'Evillage', 0.5))), ('Btown', (('Btown', 'Ccity', 5),)), ('Ccity',␣
↪(('Ccity', 'Ccity', 1), ('Ccity', 'Fstadt', 9), ('Ccity', 'Dpueblo', 3), ('Ccity',
↪'Atown', 3))), ('Dpueblo', (('Dpueblo', 'Ccity', 2), ('Dpueblo', 'Fstadt', 10))), (
↪'Evillage', (('Evillage', 'Btown', 1),)), ('Fstadt', (('Fstadt', 'Ccity', 3),))]
Path so far: [] (length 0). Search from Atown to Atown
Path so far: ['Atown'] (length 0). Search from Atown to Btown
Path so far: ['Atown'] (length 0). Search from Atown to Ccity
Path so far: ['Atown'] (length 0). Search from Atown to Atown
Path so far: ['Atown'] (length 0). Search from Atown to Evillage
```

```
Path so far: ['Atown', 'Evillage'] (length 0.5). Search from Evillage to Btown
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Ccity
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Fstadt
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Dpueblo
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Atown
Path so far: ['Atown', 'Evillage', 'Btown'] (length 1.5). Search from Btown to Ccity
Path so far: ['Atown', 'Ccity', 'Dpueblo'] (length 4). Search from Dpueblo to Ccity
Path so far: ['Atown', 'Ccity', 'Dpueblo'] (length 4). Search from Dpueblo to Fstadt
SHORTEST PATH ['Atown', 'Ccity', 'Fstadt'] Length: 10
```

Here is the OO version:

```
from SimPy.Simulation import *
""" shortestPath_SimPy_OO.py

    Finds the shortest path in a network.
    Author: Klaus Muller
"""
# Model components ------------------------


class node:
    def __init__(self):
        self.reached = 0


class searcher(Process):
    def __init__(self, graph, path, length, from_node,
                 to_node, distance, goal_node, sim):
        Process.__init__(self, sim=sim)
        self.path = path[:]
        self.length = length
        self.from_node = from_node
        self.to_node = to_node
        self.distance = distance
        self.graph = graph
        self.goal_node = goal_node

    def run(self, to_node):
        if DEMO:
            print("Path so far: {0} (length {1}). "
                  "Search from {2} to {3}".format(self.path,
                                                  self.length,
                                                  self.from_node,
                                                  to_node))
        yield hold, self, self.distance
        if not self.sim.nodes[to_node].reached:
            self.path.append(to_node)
            self.length += self.distance
            self.sim.nodes[to_node].reached = 1
            if to_node == self.goal_node:
                print("SHORTEST PATH", self.path, "Length:", self.length)
                self.sim.stopSimulation()
            else:
                for i in self.graph[to_node]:
                    s = searcher(graph=self.graph, path=self.path,
                                 length=self.length, from_node=i[0],
                                 to_node=i[1], distance=i[2],
```

```
                                goal_node=self.goal_node, sim=self.sim)
                    self.sim.activate(s, s.run(i[1]))

# Model --------------------------------


class ShortestPathModel(Simulation):
    def search(self):
        print('shortestPath_SimPy')
        self.initialize()
        self.nodes = {}
        for i in ("Atown", "Btown", "Ccity", "Dpueblo", "Evillage", "Fstadt"):
            self.nodes[i] = node()
        """ Format graph definition:
        a_graph={node_id:[(from,to,distance),
                          (from,to,distance)],
                         node_id:[ . . . ])
        """
        net = {"Atown": (("Atown", "Btown", 3.5), ("Atown", "Ccity", 1),
                         ("Atown", "Atown", 9), ("Atown", "Evillage", 0.5)),
               "Btown": (("Btown", "Ccity", 5),),
               "Ccity": (("Ccity", "Ccity", 1), ("Ccity", "Fstadt", 9),
                         ("Ccity", "Dpueblo", 3), ("Ccity", "Atown", 3)),
               "Dpueblo": (("Dpueblo", "Ccity", 2), ("Dpueblo", "Fstadt", 10)),
               "Evillage": (("Evillage", "Btown", 1),),
               "Fstadt": (("Fstadt", "Ccity", 3),)}
        if DEMO:
            print("Search for shortest path from {0} to {1} \n"
                  "in graph {2}".format("Atown",
                                        "Fstadt",
                                        sorted(net.items())))
        startup = searcher(graph=net, path=[], length=0, from_node="Atown",
                           to_node="Atown", distance=0, goal_node="Fstadt",
                           sim=self)
        self.activate(startup, startup.run("Atown"))
        self.simulate(until=10000)


# Experiment ------------------------------
DEMO = 1
ShortestPathModel().search()
```

```
shortestPath_SimPy
Search for shortest path from Atown to Fstadt
in graph [('Atown', (('Atown', 'Btown', 3.5), ('Atown', 'Ccity', 1), ('Atown', 'Atown
↪', 9), ('Atown', 'Evillage', 0.5))), ('Btown', (('Btown', 'Ccity', 5),)), ('Ccity',␣
↪(('Ccity', 'Ccity', 1), ('Ccity', 'Fstadt', 9), ('Ccity', 'Dpueblo', 3), ('Ccity',
↪'Atown', 3))), ('Dpueblo', (('Dpueblo', 'Ccity', 2), ('Dpueblo', 'Fstadt', 10))), (
↪'Evillage', (('Evillage', 'Btown', 1),)), ('Fstadt', (('Fstadt', 'Ccity', 3),))]
Path so far: [] (length 0). Search from Atown to Atown
Path so far: ['Atown'] (length 0). Search from Atown to Btown
Path so far: ['Atown'] (length 0). Search from Atown to Ccity
Path so far: ['Atown'] (length 0). Search from Atown to Atown
Path so far: ['Atown'] (length 0). Search from Atown to Evillage
Path so far: ['Atown', 'Evillage'] (length 0.5). Search from Evillage to Btown
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Ccity
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Fstadt
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Dpueblo
```

```
Path so far: ['Atown', 'Ccity'] (length 1). Search from Ccity to Atown
Path so far: ['Atown', 'Evillage', 'Btown'] (length 1.5). Search from Btown to Ccity
Path so far: ['Atown', 'Ccity', 'Dpueblo'] (length 4). Search from Dpueblo to Ccity
Path so far: ['Atown', 'Ccity', 'Dpueblo'] (length 4). Search from Dpueblo to Fstadt
SHORTEST PATH ['Atown', 'Ccity', 'Fstadt'] Length: 10
```

Machine Shop Model: Machineshop.py, Machineshop_OO.py

A workshop has n identical machines. A stream of jobs (enough to keep the machines busy) arrives. Each machine breaks down periodically. Repairs are carried out by one repairman. The repairman has other, less important tasks to perform, too. Once he starts one of those, he completes it before starting with the machine repair. The workshop works continuously.

This is an example of the use of the `interrupt()` method. (KGM)

```
from SimPy.Simulation import *
import random
"""
Machineshop Model

An example showing interrupts and priority queuing.

Scenario:

A workshop has n identical machines. A stream of jobs (enough to keep
the machines busy) arrives. Each machine breaks down
periodically. Repairs are carried out by one repairman. The repairman
has other, less important tasks to perform, too. Once he starts one of
those, he completes it before starting with the machine repair. The
workshop works continously.  """

# Model components ------------------------


class Machine(Process):
    def __init__(self, name):
        Process.__init__(self, name)
        myBreaker = Breakdown(self)
        activate(myBreaker, myBreaker.breakmachine())
        self.partsMade = 0

    def working(self):
        while True:
            yield hold, self, timePerPart()
            if self.interrupted():
                # broken down
                parttimeleft = self.interruptLeft
                yield request, self, repairman, 1
                yield hold, self, repairtime
                # repaired
                yield release, self, repairman
                yield hold, self, parttimeleft
                # part completed
                self.partsMade += 1
            else:
                # part made
```

```
                self.partsMade += 1


class Breakdown(Process):
    def __init__(self, myMachine):
        Process.__init__(self)
        self.myMachine = myMachine

    def breakmachine(self):
        while True:
            yield hold, self, timeToFailure()
            self.interrupt(self.myMachine)


class OtherJobs(Process):
    def __init__(self):
        Process.__init__(self)

    def doingJobs(self):
        while True:
            yield request, self, repairman, 0
            # starts working on jobs
            yield hold, self, jobDuration
            yield release, self, repairman


def timePerPart():
    return random.normalvariate(processingTimeMean, processingTimeSigma)


def timeToFailure():
    return random.expovariate(mean)

# Experiment data --------------------------


repairtime = 30.0           # minutes
processingTimeMean = 10.0   # minutes
processingTimeSigma = 2.0
timeToFailureMean = 300.0  # minutes
mean = 1 / timeToFailureMean  # per minute
jobDuration = 30            # minutes
nrMachines = 10
random.seed(111333555)
weeks = 4  # weeks
simTime = weeks * 24 * 60 * 7  # minutes

# Model/Experiment ------------------------------

print('Machineshop')
initialize()
repairman = Resource(capacity=1, qType=PriorityQ)
m = {}
for i in range(nrMachines):
    m[i + 1] = Machine(name="Machine {0}".format(i + 1))
    activate(m[i + 1], m[i + 1].working())
oj = OtherJobs()
activate(oj, oj.doingJobs())
```

```
simulate(until=simTime)   # minutes

# Analysis/output -------------------------

print("Machineshop results after {0} weeks".format(weeks))
for i in range(nrMachines):
    print("Machine {0}: {1}".format(i + 1, m[i + 1].partsMade))
```

```
Machineshop
Machineshop results after 4 weeks
Machine 1: 3262
Machine 2: 3353
Machine 3: 3161
Machine 4: 3200
Machine 5: 3296
Machine 6: 3139
Machine 7: 3287
Machine 8: 3248
Machine 9: 3273
Machine 10: 3217
```

Here is the OO version:

```
from SimPy.Simulation import *
import random
""" Machineshop_OO.py
Machineshop Model

An example showing interrupts and priority queuing.

Scenario:

A workshop has n identical machines. A stream of jobs (enough to keep
the machines busy) arrives. Each machine breaks down
periodically. Repairs are carried out by one repairman. The repairman
has other, less important tasks to perform, too. Once he starts one of
those, he completes it before starting with the machine repair. The
workshop works continously.  """

# Model components ------------------------


class Machine(Process):
    def __init__(self, name, sim):
        Process.__init__(self, name=name, sim=sim)
        myBreaker = Breakdown(self, sim=sim)
        sim.activate(myBreaker, myBreaker.breakmachine())
        self.partsMade = 0

    def working(self):
        while True:
            yield hold, self, timePerPart()
            if self.interrupted():
                # broken down
                parttimeleft = self.interruptLeft
                yield request, self, self.sim.repairman, 1
                yield hold, self, repairtime
                # repaired
```

```
                yield release, self, self.sim.repairman
                yield hold, self, parttimeleft
                # part completed
                self.partsMade += 1
            else:
                # part made
                self.partsMade += 1


class Breakdown(Process):
    def __init__(self, myMachine, sim):
        Process.__init__(self, sim=sim)
        self.myMachine = myMachine

    def breakmachine(self):
        while True:
            yield hold, self, timeToFailure()
            self.interrupt(self.myMachine)


class OtherJobs(Process):

    def doingJobs(self):
        while True:
            yield request, self, self.sim.repairman, 0
            # starts working on jobs
            yield hold, self, jobDuration
            yield release, self, self.sim.repairman


def timePerPart():
    return random.normalvariate(processingTimeMean, processingTimeSigma)


def timeToFailure():
    return random.expovariate(mean)

# Experiment data ------------------------


repairtime = 30.0           # minutes
processingTimeMean = 10.0   # minutes
processingTimeSigma = 2.0
timeToFailureMean = 300.0  # minutes
mean = 1 / timeToFailureMean  # per minute
jobDuration = 30            # minutes
nrMachines = 10
random.seed(111333555)
weeks = 4  # weeks
simTime = weeks * 24 * 60 * 7  # minutes

# Model


class MachineshopModel(Simulation):
    def run(self):
        print('Machineshop')
        self.initialize()
```

```
        self.repairman = Resource(capacity=1, qType=PriorityQ, sim=self)
        self.m = {}
        for i in range(nrMachines):
            self.m[i + 1] = Machine(name="Machine {0}".format(i + 1), sim=self)
            self.activate(self.m[i + 1], self.m[i + 1].working())
        oj = OtherJobs(sim=self)
        self.activate(oj, oj.doingJobs())
        self.simulate(until=simTime)  # minutes


# Experiment ------------------------------
model = MachineshopModel()
model.run()

# Analysis/output -------------------------

print("Machineshop results after {0} weeks".format(weeks))
for i in range(nrMachines):
    print("Machine {0}: {1}".format(i + 1, model.m[i + 1].partsMade))
```

```
Machineshop
Machineshop results after 4 weeks
Machine 1: 3262
Machine 2: 3353
Machine 3: 3161
Machine 4: 3200
Machine 5: 3296
Machine 6: 3139
Machine 7: 3287
Machine 8: 3248
Machine 9: 3273
Machine 10: 3217
```

Supermarket: Market.py, Market_OO.py

A supermarket checkout with multiple counters and extended Monitor objects. Written and analysed by David Mertz in an article for developerWorks (). (MM)

```
""" Market.py
Model of a supermarket.
"""
from SimPy.Simulation import *
import random
from math import sqrt


# Model components ------------------------

class Customer(Process):
    def __init__(self):
        Process.__init__(self)
        # Randomly pick how many items this customer is buying
        self.items = 1 + int(random.expovariate(1.0 / AVGITEMS))

    def checkout(self):
        start = now()           # Customer decides to check out
```

```
        yield request, self, checkout_aisle
        at_checkout = now()      # Customer gets to front of line
        waittime.tally(at_checkout - start)
        yield hold, self, self.items * ITEMTIME
        leaving = now()          # Customer completes purchase
        checkouttime.tally(leaving - at_checkout)
        yield release, self, checkout_aisle


class Customer_Factory(Process):
    def run(self):
        while 1:
            c = Customer()
            activate(c, c.checkout())
            arrival = random.expovariate(float(AVGCUST) / CLOSING)
            yield hold, self, arrival


class Monitor2(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.min, self.max = (sys.maxsize, 0)

    def tally(self, x):
        self.observe(x)
        self.min = min(self.min, x)
        self.max = max(self.max, x)

# Experiment data --------------------------


AISLES = 6          # Number of open aisles
ITEMTIME = 0.1      # Time to ring up one item
AVGITEMS = 20       # Average number of items purchased
CLOSING = 60 * 12     # Minutes from store open to store close
AVGCUST = 1500      # Average number of daily customers
RUNS = 8            # Number of times to run the simulation
SEED = 111333555    # seed value for random numbers


# Model/Experiment ------------------------------

random.seed(SEED)
print('Market')
for run in range(RUNS):
    waittime = Monitor2()
    checkouttime = Monitor2()
    checkout_aisle = Resource(AISLES)
    initialize()
    cf = Customer_Factory()
    activate(cf, cf.run(), 0.0)
    simulate(until=CLOSING)
    FMT = "Waiting time average: {0:.1f} (std dev {1:.1f}, maximum {2:.1f})"
    print(FMT.format(waittime.mean(), sqrt(waittime.var()), waittime.max))

# Analysis/output --------------------------

print('AISLES:', AISLES, '  ITEM TIME:', ITEMTIME)
```

```
Market
Waiting time average: 0.5 (std dev 1.0, maximum 6.2)
Waiting time average: 0.3 (std dev 0.7, maximum 3.8)
Waiting time average: 0.3 (std dev 0.5, maximum 3.1)
Waiting time average: 0.3 (std dev 0.7, maximum 4.1)
Waiting time average: 0.3 (std dev 0.6, maximum 3.5)
Waiting time average: 0.3 (std dev 0.6, maximum 4.3)
Waiting time average: 0.4 (std dev 0.9, maximum 5.6)
Waiting time average: 0.2 (std dev 0.5, maximum 3.7)
AISLES: 6   ITEM TIME: 0.1
```

Here is the OO version:

```
""" Market_OO.py
Model of a supermarket.
"""
from SimPy.Simulation import *
import random
from math import sqrt


# Model components ------------------------

class Customer(Process):
    def __init__(self, sim):
        Process.__init__(self, sim=sim)
        # Randomly pick how many items this customer is buying
        self.items = 1 + int(random.expovariate(1.0 / AVGITEMS))

    def checkout(self):
        start = self.sim.now()           # Customer decides to check out
        yield request, self, self.sim.checkout_aisle
        at_checkout = self.sim.now()     # Customer gets to front of line
        self.sim.waittime.tally(at_checkout - start)
        yield hold, self, self.items * ITEMTIME
        leaving = self.sim.now()         # Customer completes purchase
        self.sim.checkouttime.tally(leaving - at_checkout)
        yield release, self, self.sim.checkout_aisle


class Customer_Factory(Process):
    def run(self):
        while 1:
            c = Customer(sim=self.sim)
            self.sim.activate(c, c.checkout())
            arrival = random.expovariate(float(AVGCUST) / CLOSING)
            yield hold, self, arrival


class Monitor2(Monitor):
    def __init__(self, sim):
        Monitor.__init__(self, sim=sim)
        self.min, self.max = (sys.maxsize, 0)

    def tally(self, x):
        self.observe(x)
        self.min = min(self.min, x)
        self.max = max(self.max, x)
```

```
# Experiment data ------------------------


AISLES = 6          # Number of open aisles
ITEMTIME = 0.1      # Time to ring up one item
AVGITEMS = 20       # Average number of items purchased
CLOSING = 60 * 12     # Minutes from store open to store close
AVGCUST = 1500      # Average number of daily customers
RUNS = 8            # Number of times to run the simulation
SEED = 111333555    # seed value for random numbers


# Model
class MarketModel(Simulation):
    def runs(self):

        random.seed(SEED)
        print('Market')
        for run in range(RUNS):
            self.initialize()
            self.waittime = Monitor2(sim=self)
            self.checkouttime = Monitor2(sim=self)
            self.checkout_aisle = Resource(capacity=AISLES, sim=self)

            cf = Customer_Factory(sim=self)
            self.activate(cf, cf.run(), 0.0)
            self.simulate(until=CLOSING)
            # Analysis/output -------------
            FMT = ("Waiting time average: {0:.1f} "
                   "(std dev {1:.1f}, maximum {2:.1f})")
            print(FMT.format(self.waittime.mean(),
                             sqrt(self.waittime.var()),
                             self.waittime.max))


# Experiment -----------------------------
MarketModel().runs()
print('AISLES:', AISLES, '  ITEM TIME:', ITEMTIME)
```

```
Market
Waiting time average: 0.5 (std dev 1.0, maximum 6.2)
Waiting time average: 0.3 (std dev 0.7, maximum 3.8)
Waiting time average: 0.3 (std dev 0.5, maximum 3.1)
Waiting time average: 0.3 (std dev 0.7, maximum 4.1)
Waiting time average: 0.3 (std dev 0.6, maximum 3.5)
Waiting time average: 0.3 (std dev 0.6, maximum 4.3)
Waiting time average: 0.4 (std dev 0.9, maximum 5.6)
Waiting time average: 0.2 (std dev 0.5, maximum 3.7)
AISLES: 6   ITEM TIME: 0.1
```

Movie Theatre Ticket Counter: Movie_renege.py, Movie_renege_OO.py

Use of reneging (compound `yield request`) constructs for reneging at occurrence of an event. Scenario is a movie ticket counter with a limited number of tickets for three movies (next show only). When a movie is sold out, all people waiting to buy ticket for that movie renege (leave queue).

```
"""
Movie_renege

Demo program to show event-based reneging by
'yield (request,self,res),(waitevent,self,evt)'.

Scenario:
A movie theatre has one ticket counter selling tickets for
three movies (next show only). When a movie is sold out, all
people waiting to buy ticket for that movie renege (leave queue).
"""
from SimPy.Simulation import *
from random import *

# Model components ------------------------


class MovieGoer(Process):
    def getTickets(self, whichMovie, nrTickets):
        yield ((request, self, ticketCounter),
               (waitevent, self, soldOut[whichMovie]))
        if self.acquired(ticketCounter):
            if available[whichMovie] >= nrTickets:
                available[whichMovie] -= nrTickets
                if available[whichMovie] < 2:
                    soldOut[whichMovie].signal()
                    whenSoldOut[whichMovie] = now()
                    available[whichMovie] = 0
                yield hold, self, 1
            else:
                yield hold, self, 0.5
            yield release, self, ticketCounter
        else:
            nrRenegers[whichMovie] += 1


class CustomerArrivals(Process):
    def traffic(self):
        while now() < 120:
            yield hold, self, expovariate(1 / 0.5)
            movieChoice = choice(movies)
            nrTickets = randint(1, 6)
            if available[movieChoice]:
                m = MovieGoer()
                activate(m, m.getTickets(
                    whichMovie=movieChoice, nrTickets=nrTickets))


# Experiment data -------------------------
seed(111333555)
movies = ["Gone with the Windows", "Hard Core Dump", "Modern CPU Times"]

available = {}
soldOut = {}
nrRenegers = {}
whenSoldOut = {}
for film in movies:
    available[film] = 50
```

```
    soldOut[film] = SimEvent(film)
    nrRenegers[film] = 0
ticketCounter = Resource(capacity=1)

# Model/Experiment ------------------------------
print('Movie_renege')
initialize()
c = CustomerArrivals()
activate(c, c.traffic())
simulate(until=120)

for f in movies:
    if soldOut[f]:
        FMT = ("Movie '{0}' sold out {1:.0f} minutes after ticket counter "
               "opening.")
        print(FMT.format(f, int(whenSoldOut[f])))
        FMT = "\tNr people leaving queue when film '{0}' sold out: {1}"
        print(FMT.format(f, nrRenegers[f]))
```

```
Movie_renege
Movie 'Gone with the Windows' sold out 41 minutes after ticket counter opening.
        Nr people leaving queue when film 'Gone with the Windows' sold out: 9
Movie 'Hard Core Dump' sold out 37 minutes after ticket counter opening.
        Nr people leaving queue when film 'Hard Core Dump' sold out: 8
Movie 'Modern CPU Times' sold out 33 minutes after ticket counter opening.
        Nr people leaving queue when film 'Modern CPU Times' sold out: 9
```

Here is the OO version:

```
"""Movie_renege.py

Demo program to show event-based reneging by
'yield (request,self,res),(waitevent,self,evt)'.

Scenario:
A movie theatre has one ticket counter selling tickets for
three movies (next show only). When a movie is sold out, all
people waiting to buy ticket for that movie renege (leave queue).
"""
from SimPy.Simulation import *
from random import *

# Model components ------------------------


class MovieGoer(Process):
    def getTickets(self, whichMovie, nrTickets):
        yield ((request, self, self.sim.ticketCounter),
               (waitevent, self, self.sim.soldOut[whichMovie]))
        if self.acquired(self.sim.ticketCounter):
            if self.sim.available[whichMovie] >= nrTickets:
                self.sim.available[whichMovie] -= nrTickets
                if self.sim.available[whichMovie] < 2:
                    self.sim.soldOut[whichMovie].signal()
                    self.sim.whenSoldOut[whichMovie] = self.sim.now()
                    self.sim.available[whichMovie] = 0
                yield hold, self, 1
            else:
```

```
                yield hold, self, 0.5
            yield release, self, self.sim.ticketCounter
        else:
            self.sim.nrRenegers[whichMovie] += 1


class CustomerArrivals(Process):
    def traffic(self):
        while self.sim.now() < 120:
            yield hold, self, expovariate(1 / 0.5)
            movieChoice = choice(movies)
            nrTickets = randint(1, 6)
            if self.sim.available[movieChoice]:
                m = MovieGoer(sim=self.sim)
                self.sim.activate(m, m.getTickets(
                    whichMovie=movieChoice, nrTickets=nrTickets))


# Experiment data -------------------------
seed(111333555)
movies = ["Gone with the Windows", "Hard Core Dump", "Modern CPU Times"]

# Model -----------------------------------


class MovieRenegeModel(Simulation):
    def run(self):
        print('Movie_renege')
        self.initialize()
        self.available = {}
        self.soldOut = {}
        self.nrRenegers = {}
        self.whenSoldOut = {}
        for film in movies:
            self.available[film] = 50
            self.soldOut[film] = SimEvent(film, sim=self)
            self.nrRenegers[film] = 0
        self.ticketCounter = Resource(capacity=1, sim=self)
        c = CustomerArrivals(sim=self)
        self.activate(c, c.traffic())
        self.simulate(until=120)


# Experiment ------------------------------
model = MovieRenegeModel()
model.run()

# Analysis/output -------------------------
for f in movies:
    if model.soldOut[f]:
        FMT = ("Movie '{0}' sold out {1:.0f} minutes after ticket counter "
               "opening.")
        print(FMT.format(f, int(model.whenSoldOut[f])))
        FMT = "\tNr people leaving queue when film '{0}' sold out: {1}"
        print(FMT.format(f, model.nrRenegers[f]))
```

```
Movie_renege
Movie 'Gone with the Windows' sold out 41 minutes after ticket counter opening.
```

```
        Nr people leaving queue when film 'Gone with the Windows' sold out: 9
Movie 'Hard Core Dump' sold out 37 minutes after ticket counter opening.
        Nr people leaving queue when film 'Hard Core Dump' sold out: 8
Movie 'Modern CPU Times' sold out 33 minutes after ticket counter opening.
        Nr people leaving queue when film 'Modern CPU Times' sold out: 9
```

Workers Sharing Tools, waitUntil: needResources.py, needResources_OO.py

Demo of `waitUntil` capability. It simulates three workers each requiring a set of tools to do their jobs. Tools are shared, scarce resources for which they compete.

```
from SimPy.Simulation import *
import random

"""
needResources.py

Demo of waitUntil capability.

Scenario:
Three workers require sets of tools to do their jobs. Tools are shared, scarce
resources for which they compete.
"""


class Worker(Process):
    def work(self, heNeeds=[]):
        def workerNeeds():
            for item in heNeeds:
                if item.n == 0:
                    return False
            return True

        while now() < 8 * 60:
            yield waituntil, self, workerNeeds
            for item in heNeeds:
                yield request, self, item
            print("{0} {1} has {2} and starts job"
                  .format(now(),
                          self.name,
                          [x.name for x in heNeeds]))
            yield hold, self, random.uniform(10, 30)
            for item in heNeeds:
                yield release, self, item
            yield hold, self, 2  # rest


random.seed(111333555)
print('needResources')
initialize()
brush = Resource(capacity=1, name="brush")
ladder = Resource(capacity=2, name="ladder")
hammer = Resource(capacity=1, name="hammer")
saw = Resource(capacity=1, name="saw")
painter = Worker("painter")
activate(painter, painter.work([brush, ladder]))
```

```
roofer = Worker("roofer")
activate(roofer, roofer.work([hammer, ladder, ladder]))
treeguy = Worker("treeguy")
activate(treeguy, treeguy.work([saw, ladder]))
print(simulate(until=9 * 60))
```

```
needResources
0 painter has ['brush', 'ladder'] and starts job
16.4985835442738 roofer has ['hammer', 'ladder', 'ladder'] and starts job
39.41544751014284 treeguy has ['saw', 'ladder'] and starts job
39.41544751014284 painter has ['brush', 'ladder'] and starts job
56.801814464340836 roofer has ['hammer', 'ladder', 'ladder'] and starts job
75.30359743269759 painter has ['brush', 'ladder'] and starts job
75.30359743269759 treeguy has ['saw', 'ladder'] and starts job
103.081870230719 roofer has ['hammer', 'ladder', 'ladder'] and starts job
118.17856121225341 painter has ['brush', 'ladder'] and starts job
118.17856121225341 treeguy has ['saw', 'ladder'] and starts job
143.25197748702192 roofer has ['hammer', 'ladder', 'ladder'] and starts job
170.86533741129648 treeguy has ['saw', 'ladder'] and starts job
170.86533741129648 painter has ['brush', 'ladder'] and starts job
196.3748629530008 roofer has ['hammer', 'ladder', 'ladder'] and starts job
217.8426181665415 treeguy has ['saw', 'ladder'] and starts job
217.8426181665415 painter has ['brush', 'ladder'] and starts job
244.3699827271992 roofer has ['hammer', 'ladder', 'ladder'] and starts job
269.8696310798061 painter has ['brush', 'ladder'] and starts job
269.8696310798061 treeguy has ['saw', 'ladder'] and starts job
298.28127810116723 roofer has ['hammer', 'ladder', 'ladder'] and starts job
316.83616616186885 treeguy has ['saw', 'ladder'] and starts job
316.83616616186885 painter has ['brush', 'ladder'] and starts job
345.4248822185373 roofer has ['hammer', 'ladder', 'ladder'] and starts job
359.2558219106454 painter has ['brush', 'ladder'] and starts job
359.2558219106454 treeguy has ['saw', 'ladder'] and starts job
385.7096335445383 roofer has ['hammer', 'ladder', 'ladder'] and starts job
407.43697134395245 painter has ['brush', 'ladder'] and starts job
407.43697134395245 treeguy has ['saw', 'ladder'] and starts job
427.5649730427561 roofer has ['hammer', 'ladder', 'ladder'] and starts job
451.6157857370266 painter has ['brush', 'ladder'] and starts job
451.6157857370266 treeguy has ['saw', 'ladder'] and starts job
472.97306394134876 roofer has ['hammer', 'ladder', 'ladder'] and starts job
492.3976188422561 treeguy has ['saw', 'ladder'] and starts job
492.3976188422561 painter has ['brush', 'ladder'] and starts job
SimPy: No more events at time 515.0226394595458
```

Here is the OO version:

```
from SimPy.Simulation import *
import random

"""
needResources.py

Demo of waitUntil capability.

Scenario:
Three workers require sets of tools to do their jobs. Tools are shared, scarce
resources for which they compete.
"""
```

```
# Model components ------------------------


class Worker(Process):
    def work(self, heNeeds=[]):
        def workerNeeds():
            for item in heNeeds:
                if item.n == 0:
                    return False
            return True

        while self.sim.now() < 8 * 60:
            yield waituntil, self, workerNeeds
            for item in heNeeds:
                yield request, self, item
            print("{0} {1} has {2} and starts job"
                  .format(self.sim.now(),
                          self.name,
                          [x.name for x in heNeeds]))
            yield hold, self, random.uniform(10, 30)
            for item in heNeeds:
                yield release, self, item
            yield hold, self, 2  # rest


# Model -----------------------------------
class NeedResourcesModel(Simulation):
    def run(self):
        print('needResources')
        self.initialize()
        brush = Resource(capacity=1, name="brush", sim=self)
        ladder = Resource(capacity=2, name="ladder", sim=self)
        hammer = Resource(capacity=1, name="hammer", sim=self)
        saw = Resource(capacity=1, name="saw", sim=self)
        painter = Worker("painter", sim=self)
        self.activate(painter, painter.work([brush, ladder]))
        roofer = Worker("roofer", sim=self)
        self.activate(roofer, roofer.work([hammer, ladder, ladder]))
        treeguy = Worker("treeguy", sim=self)
        self.activate(treeguy, treeguy.work([saw, ladder]))
        print(self.simulate(until=9 * 60))


# Experiment data -------------------------
SEED = 111333555

# Experiment ------------------------------
random.seed(SEED)
NeedResourcesModel().run()
```

```
needResources
0 painter has ['brush', 'ladder'] and starts job
16.4985835442738 roofer has ['hammer', 'ladder', 'ladder'] and starts job
39.41544751014284 treeguy has ['saw', 'ladder'] and starts job
39.41544751014284 painter has ['brush', 'ladder'] and starts job
56.801814464340836 roofer has ['hammer', 'ladder', 'ladder'] and starts job
75.30359743269759 painter has ['brush', 'ladder'] and starts job
75.30359743269759 treeguy has ['saw', 'ladder'] and starts job
```

```
103.081870230719 roofer has ['hammer', 'ladder', 'ladder'] and starts job
118.17856121225341 painter has ['brush', 'ladder'] and starts job
118.17856121225341 treeguy has ['saw', 'ladder'] and starts job
143.25197748702192 roofer has ['hammer', 'ladder', 'ladder'] and starts job
170.86533741129648 treeguy has ['saw', 'ladder'] and starts job
170.86533741129648 painter has ['brush', 'ladder'] and starts job
196.3748629530008 roofer has ['hammer', 'ladder', 'ladder'] and starts job
217.8426181665415 treeguy has ['saw', 'ladder'] and starts job
217.8426181665415 painter has ['brush', 'ladder'] and starts job
244.3699827271992 roofer has ['hammer', 'ladder', 'ladder'] and starts job
269.8696310798061 painter has ['brush', 'ladder'] and starts job
269.8696310798061 treeguy has ['saw', 'ladder'] and starts job
298.28127810116723 roofer has ['hammer', 'ladder', 'ladder'] and starts job
316.83616616186885 treeguy has ['saw', 'ladder'] and starts job
316.83616616186885 painter has ['brush', 'ladder'] and starts job
345.4248822185373 roofer has ['hammer', 'ladder', 'ladder'] and starts job
359.2558219106454 painter has ['brush', 'ladder'] and starts job
359.2558219106454 treeguy has ['saw', 'ladder'] and starts job
385.7096335445383 roofer has ['hammer', 'ladder', 'ladder'] and starts job
407.43697134395245 painter has ['brush', 'ladder'] and starts job
407.43697134395245 treeguy has ['saw', 'ladder'] and starts job
427.5649730427561 roofer has ['hammer', 'ladder', 'ladder'] and starts job
451.6157857370266 painter has ['brush', 'ladder'] and starts job
451.6157857370266 treeguy has ['saw', 'ladder'] and starts job
472.97306394134876 roofer has ['hammer', 'ladder', 'ladder'] and starts job
492.3976188422561 treeguy has ['saw', 'ladder'] and starts job
492.3976188422561 painter has ['brush', 'ladder'] and starts job
SimPy: No more events at time 515.0226394595458
```

Widget Factory: SimPy_worker_extend.py, SimPy_worker_extend_OO.py

Factory making widgets with queues for machines. (MM)

```
from SimPy.Simulation import *
from random import uniform, seed


def theTime(time):

    hrs = int(time / 60)
    min = int(time - hrs * 60)
    return "{0}:{1}".format(str.zfill(str(hrs), 2), str.zfill(str(min), 2))


class worker(Process):
    def __init__(self, id):
        Process.__init__(self)
        self.id = id
        self.output = 0
        self.idle = 0
        self.total_idle = 0

    def working_day(self, foobar):
        print("{0} Worker {1} arrives in factory".format(
            theTime(now()), self.id))
        while now() < 17 * 60:  # work till 5 pm
```

```
            yield hold, self, uniform(3, 10)
            # print("{0} Widget completed".format(theTime(now())))
            foobar.queue.append(self)
            if foobar.idle:
                reactivate(foobar)
            else:
                self.idle = 1  # worker has to wait for foobar service
                start_idle = now()
                # print("{0} Worker {1} queuing for foobar machine"
                #       .format(theTime(now()),self.id))
            yield passivate, self  # waiting and foobar service
            self.output += 1
            if self.idle:
                self.total_idle += now() - start_idle
            self.idle = 0
        print("{0} {1} goes home, having built {2:d} widgets today.".format(
            theTime(now()), self.id, self.output))
        print("Worker {0} was idle waiting for foobar machine for "
              "{1:3.1f} hours".format(self.id, self.total_idle / 60))


class foobar_machine(Process):
    def __init__(self):
        Process.__init__(self)
        self.queue = []
        self.idle = 1

    def foobar_Process(self):
        yield passivate, self
        while 1:
            while len(self.queue) > 0:
                self.idle = 0
                yield hold, self, 3
                served = self.queue.pop(0)
                reactivate(served)
            self.idle = 1
            yield passivate, self


seed(111333555)
print('SimPy_worker_extend')
initialize()
foo = foobar_machine()
activate(foo, foo.foobar_Process(), delay=0)
john = worker("John")
activate(john, john.working_day(foo), at=510)  # start at 8:30 am
eve = worker("Eve")
activate(eve, eve.working_day(foo), at=510)
simulate(until=18 * 60)
# scheduler(till=18*60) #run simulation from midnight till 6 pm
```

```
SimPy_worker_extend
08:30 Worker John arrives in factory
08:30 Worker Eve arrives in factory
17:03 Eve goes home, having built 52 widgets today.
Worker Eve was idle waiting for foobar machine for 1.3 hours
17:09 John goes home, having built 52 widgets today.
Worker John was idle waiting for foobar machine for 0.8 hours
```

Here is the OO version:

```
from SimPy.Simulation import *
from random import uniform, seed

# Model components ------------------------


def theTime(time):

    hrs = int(time / 60)
    min = int(time - hrs * 60)
    return "{0}:{1}".format(str.zfill(str(hrs), 2), str.zfill(str(min), 2))


class worker(Process):
    def __init__(self, id, sim):
        Process.__init__(self, sim=sim)
        self.id = id
        self.output = 0
        self.idle = 0
        self.total_idle = 0

    def working_day(self, foobar):
        print("{0} Worker {1} arrives in factory".format(
            theTime(self.sim.now()), self.id))
        while self.sim.now() < 17 * 60:  # work till 5 pm
            yield hold, self, uniform(3, 10)
            # print("{0} Widget completed".format(theTime(now())))
            foobar.queue.append(self)
            if foobar.idle:
                self.sim.reactivate(foobar)
            else:
                self.idle = 1  # worker has to wait for foobar service
                start_idle = self.sim.now()
                # print("{0} Worker {1} queuing for foobar machine"
                #       .format(theTime(now()),self.id))
            yield passivate, self  # waiting and foobar service
            self.output += 1
            if self.idle:
                self.total_idle += self.sim.now() - start_idle
            self.idle = 0
        print("{0} {1} goes home, having built {2} widgets today.".format(
            theTime(self.sim.now()), self.id, self.output))
        print("Worker {0} was idle waiting for foobar machine for "
              "{1:3.1f} hours".format(self.id, self.total_idle / 60))


class foobar_machine(Process):
    def __init__(self, sim):
        Process.__init__(self, sim=sim)
        self.queue = []
        self.idle = 1

    def foobar_Process(self):
        yield passivate, self
        while 1:
```

```
            while len(self.queue) > 0:
                self.idle = 0
                yield hold, self, 3
                served = self.queue.pop(0)
                self.sim.reactivate(served)
            self.idle = 1
            yield passivate, self

# Model ------------------------------------


class SimPy_Worker_Extend_Model(Simulation):
    def run(self):
        print('SimPy_worker_extend')
        self.initialize()
        foo = foobar_machine(sim=self)
        self.activate(foo, foo.foobar_Process(), delay=0)
        john = worker("John", sim=self)
        self.activate(john, john.working_day(foo), at=510)  # start at 8:30 am
        eve = worker("Eve", sim=self)
        self.activate(eve, eve.working_day(foo), at=510)
        self.simulate(until=18 * 60)  # run simulation from midnight till 6 pm


# Expriment -------------------------------
seed(111333555)
SimPy_Worker_Extend_Model().run()
```

```
SimPy_worker_extend
08:30 Worker John arrives in factory
08:30 Worker Eve arrives in factory
17:03 Eve goes home, having built 52 widgets today.
Worker Eve was idle waiting for foobar machine for 1.3 hours
17:09 John goes home, having built 52 widgets today.
Worker John was idle waiting for foobar machine for 0.8 hours
```

Widget Packing Machine: WidgetPacking.py, WidgetPacking_OO.py

Using buffers for producer/consumer scenarios. Scenario is a group of widget producing machines and a widget packer, synchronised via a buffer. Two models are shown: the first uses a Level for buffering non-distinguishable items (widgets), and the second a Store for distinguishable items (widgets of different weight).

```
"""WidgetPacking.py
Scenario:
In a factory, nProd widget-making machines produce widgets with a weight
widgWeight (uniformly distributed [widgWeight-dWeight..widgWeight+dWeight])
on average every tMake minutes (uniform distribution
[tmake-deltaT..tMake+deltaT]).A widget-packing machine packs widgets in tPack
minutes into packages. Simulate for simTime minutes.

Model 1:
The widget-packer packs nWidgets into a package.

Model 2:
The widget-packer packs widgets into a package with package weight not
to exceed packMax.
```

```
"""
from SimPy.Simulation import *
import random

# Experiment data ------------------------
nProd = 2  # widget-making machines
widgWeight = 20  # kilogrammes
dWeight = 3  # kilogrammes
tMake = 2  # minutes
deltaT = 0.75  # minutes
tPack = 0.25  # minutes per idget
initialSeed = 1234567  # for random number stream
simTime = 100  # minutes

print('WidgetPacking')
##############################################################
#
# Model 1
#
##############################################################
# Data
nWidgets = 6  # widgets per package


class Widget:
    def __init__(self, weight):
        self.weight = weight


class WidgetMakerN(Process):
    """Produces widgets"""

    def make(self, buffer):
        while True:
            yield hold, self, r.uniform(tMake - deltaT, tMake + deltaT)
            yield put, self, buffer, 1          # buffer 1 widget


class WidgetPackerN(Process):
    """Packs a number of widgets into a package"""

    def pack(self, buffer):
        while True:
            for i in range(nWidgets):
                yield get, self, buffer, 1  # get widget
                yield hold, self, tPack  # pack it
            print("{0}: package completed".format(now()))


print("Model 1: pack {0} widgets per package".format(nWidgets))
initialize()
r = random.Random()
r.seed(initialSeed)
wBuffer = Level(name="WidgetBuffer", capacity=500)
for i in range(nProd):
    wm = WidgetMakerN(name="WidgetMaker{0}".format(i))
    activate(wm, wm.make(wBuffer))
wp = WidgetPackerN(name="WidgetPacker")
```

```
activate(wp, wp.pack(wBuffer))
simulate(until=simTime)

################################################################
#
# Model 2
#
################################################################
# Data
packMax = 120  # kilogrammes per package (max)


class WidgetMakerW(Process):
    """Produces widgets"""

    def make(self, buffer):
        while True:
            yield hold, self, r.uniform(tMake - deltaT, tMake + deltaT)
            widgetWeight = r.uniform(
                widgWeight - dWeight, widgWeight + dWeight)
            # buffer widget
            yield put, self, buffer, [Widget(weight=widgetWeight)]


class WidgetPackerW(Process):
    """Packs a number of widgets into a package"""

    def pack(self, buffer):
        weightLeft = 0
        while True:
            packWeight = weightLeft  # pack a widget which did not fit
            # into previous package
            weightLeft = 0
            while True:
                yield get, self, buffer, 1  # get widget
                weightReceived = self.got[0].weight
                if weightReceived + packWeight <= packMax:
                    yield hold, self, tPack  # pack it
                    packWeight += weightReceived
                else:
                    weightLeft = weightReceived  # for next package
                    break
                print("{0}: package completed. Weight= {1:6.2f} kg".format(
                    now(), packWeight))


print(
    "\nModel 2: pack widgets up to max package weight of {0}".format(packMax))
initialize()
r = random.Random()
r.seed(initialSeed)
wBuffer = Store(name="WidgetBuffer", capacity=500)
for i in range(nProd):
    wm = WidgetMakerW(name="WidgetMaker{0}".format(i))
    activate(wm, wm.make(wBuffer))
wp = WidgetPackerW(name="WidgetPacker")
activate(wp, wp.pack(wBuffer))
simulate(until=simTime)
```

```
WidgetPacking
Model 1: pack 6 widgets per package
6.925710569651607: package completed
12.932859800562765: package completed
19.296609507200795: package completed
25.827518016011588: package completed
32.28961167417064: package completed
38.27653780313223: package completed
43.516005487393166: package completed
49.84136801349117: package completed
56.02183282676389: package completed
61.22427700160065: package completed
66.78084592723073: package completed
73.45267491487972: package completed
78.52559623726368: package completed
84.36238918070774: package completed
91.00334807967428: package completed
96.25112005144973: package completed

Model 2: pack widgets up to max package weight of 120
1.8535993255800536: package completed. Weight=  20.49 kg
2.9447148714144333: package completed. Weight=  43.29 kg
3.443679099583584: package completed. Weight=  60.73 kg
5.111979202993875: package completed. Weight=  81.36 kg
5.495036360384358: package completed. Weight= 103.32 kg
7.53958534514355: package completed. Weight=  43.79 kg
9.454524267565933: package completed. Weight=  62.32 kg
9.704524267565933: package completed. Weight=  82.07 kg
11.223206345291937: package completed. Weight= 103.55 kg
13.534262617168128: package completed. Weight=  41.81 kg
13.930945836966112: package completed. Weight=  62.71 kg
15.109021173114078: package completed. Weight=  82.73 kg
15.9312940569179: package completed. Weight= 103.41 kg
18.483473187078456: package completed. Weight=  37.07 kg
19.489762737880447: package completed. Weight=  56.79 kg
20.019035270021426: package completed. Weight=  77.23 kg
21.516971139857024: package completed. Weight=  96.93 kg
21.922330284736397: package completed. Weight= 116.85 kg
24.14512998963336: package completed. Weight=  37.89 kg
24.39877824367329: package completed. Weight=  55.50 kg
26.67484997458752: package completed. Weight=  78.46 kg
26.92484997458752: package completed. Weight= 100.35 kg
28.1664758950799: package completed. Weight= 117.54 kg
29.69269527817336: package completed. Weight=  40.05 kg
31.039995345892564: package completed. Weight=  60.53 kg
31.416541860477725: package completed. Weight=  77.63 kg
32.41760810512986: package completed. Weight= 100.53 kg
32.95461868723966: package completed. Weight= 118.77 kg
35.03241015535913: package completed. Weight=  40.09 kg
37.13500099371781: package completed. Weight=  59.57 kg
37.54993412593156: package completed. Weight=  76.58 kg
39.30652910593313: package completed. Weight=  95.59 kg
39.55652910593313: package completed. Weight= 116.49 kg
42.0420842938092: package completed. Weight=  42.24 kg
42.2920842938092: package completed. Weight=  63.58 kg
43.65741443896232: package completed. Weight=  85.90 kg
43.90741443896232: package completed. Weight= 106.78 kg
46.0951882462178: package completed. Weight=  37.20 kg
```

```
47.438300452780354: package completed. Weight=  58.47 kg
47.688300452780354: package completed. Weight=  79.33 kg
49.40690976731008: package completed. Weight= 100.51 kg
50.09589376227701: package completed. Weight= 117.80 kg
52.015927666123616: package completed. Weight=  37.47 kg
53.308294328826435: package completed. Weight=  58.78 kg
54.03538112053091: package completed. Weight=  81.10 kg
55.13265226734849: package completed. Weight=  98.94 kg
56.54061780546726: package completed. Weight=  39.67 kg
58.42284478038792: package completed. Weight=  61.78 kg
58.67284478038792: package completed. Weight=  80.36 kg
60.363861577200076: package completed. Weight= 102.50 kg
62.30054431149251: package completed. Weight=  38.39 kg
62.72667501529395: package completed. Weight=  58.21 kg
64.04176585171321: package completed. Weight=  75.31 kg
64.78871826102055: package completed. Weight=  94.49 kg
65.5200550350699: package completed. Weight= 113.54 kg
67.70604486309713: package completed. Weight=  41.53 kg
69.2327615315688: package completed. Weight=  60.57 kg
69.64556242167215: package completed. Weight=  82.52 kg
70.56237737140228: package completed. Weight= 104.26 kg
72.1477929466095: package completed. Weight=  41.62 kg
72.98432452690808: package completed. Weight=  63.88 kg
73.5826902990221: package completed. Weight=  81.87 kg
75.05025424304776: package completed. Weight= 101.04 kg
77.26048662562239: package completed. Weight=  40.26 kg
77.71710036932708: package completed. Weight=  61.69 kg
79.75353580124: package completed. Weight=  80.72 kg
80.12811805319318: package completed. Weight= 100.29 kg
82.4142311699317: package completed. Weight=  39.35 kg
83.68305885096552: package completed. Weight=  57.13 kg
83.93305885096552: package completed. Weight=  74.55 kg
85.85481524806448: package completed. Weight=  93.29 kg
86.13093377784419: package completed. Weight= 115.05 kg
87.86043544925184: package completed. Weight=  41.13 kg
89.46190021907331: package completed. Weight=  63.30 kg
90.40626417525424: package completed. Weight=  82.59 kg
91.19235409838856: package completed. Weight= 105.00 kg
92.48486291641056: package completed. Weight=  37.14 kg
93.79148753229941: package completed. Weight=  56.83 kg
94.19510234363693: package completed. Weight=  76.57 kg
96.09493981469757: package completed. Weight=  94.85 kg
96.82617211079959: package completed. Weight= 114.58 kg
98.82720299189798: package completed. Weight=  39.12 kg
99.94853712506526: package completed. Weight=  58.33 kg
```

Here is the OO version:

```
"""WidgetPacking_OO.py
Scenario:
In a factory, nProd widget-making machines produce widgets with a weight
widgWeight (uniformly distributed [widgWeight-dWeight..widgWeight+dWeight])
on average every tMake minutes (uniform distribution
[tmake-deltaT..tMake+deltaT]).A widget-packing machine packs widgets in tPack
minutes into packages. Simulate for simTime minutes.

Model 1:
The widget-packer packs nWidgets into a package.
```

```
Model 2:
The widget-packer packs widgets into a package with package weight not
to exceed packMax.
"""
from SimPy.Simulation import *
import random

# Experiment data --------------------------
nProd = 2  # widget-making machines
widgWeight = 20  # kilogrammes
dWeight = 3  # kilogrammes
tMake = 2  # minutes
deltaT = 0.75  # minutes
tPack = 0.25  # minutes per idget
initialSeed = 1234567  # for random number stream
simTime = 100  # minutes

print('WidgetPacking')

# Model 1 components -----------------------

# Data
nWidgets = 6  # widgets per package


class Widget:
    def __init__(self, weight):
        self.weight = weight


class WidgetMakerN(Process):
    def make(self, buffer):
        while True:
            yield hold, self, self.sim.r.uniform(tMake - deltaT,
                                                 tMake + deltaT)
            yield put, self, buffer, 1          # buffer 1 widget


class WidgetPackerN(Process):
    """Packs a number of widgets into a package"""

    def pack(self, buffer):
        while True:
            for i in range(nWidgets):
                yield get, self, buffer, 1  # get widget
                yield hold, self, tPack  # pack it
            print("{0}: package completed".format(self.sim.now()))

# Model 1 ---------------------------------


class WidgetPackingModel1(Simulation):
    def run(self):
        print("Model 1: pack {0} widgets per package".format(nWidgets))
        self.initialize()
        self.r = random.Random()
        self.r.seed(initialSeed)
```

```
        wBuffer = Level(name="WidgetBuffer", capacity=500, sim=self)
        for i in range(nProd):
            wm = WidgetMakerN(name="WidgetMaker{0}".format(i), sim=self)
            self.activate(wm, wm.make(wBuffer))
        wp = WidgetPackerN(name="WidgetPacker", sim=self)
        self.activate(wp, wp.pack(wBuffer))
        self.simulate(until=simTime)


# Experiment ------------------------------
WidgetPackingModel1().run()

# Model 2 components ----------------------

# Data
packMax = 120  # kilogrammes per package (max)


class WidgetMakerW(Process):
    """Produces widgets"""

    def make(self, buffer):
        while True:
            yield hold, self, self.sim.r.uniform(tMake - deltaT,
                                                 tMake + deltaT)
            widgetWeight = self.sim.r.uniform(
                widgWeight - dWeight, widgWeight + dWeight)
            # buffer widget
            yield put, self, buffer, [Widget(weight=widgetWeight)]


class WidgetPackerW(Process):
    """Packs a number of widgets into a package"""

    def pack(self, buffer):
        weightLeft = 0
        while True:
            packWeight = weightLeft  # pack a widget which did not fit
            # into previous package
            weightLeft = 0
            while True:
                yield get, self, buffer, 1  # get widget
                weightReceived = self.got[0].weight
                if weightReceived + packWeight <= packMax:
                    yield hold, self, tPack  # pack it
                    packWeight += weightReceived
                else:
                    weightLeft = weightReceived  # for next package
                    break
                print("{0}: package completed. Weight= {1:6.2f} kg".format(
                    self.sim.now(), packWeight))

# Model 2 ---------------------------------


class WidgetPackingModel2(Simulation):
    def run(self):
        print(
```

```
            "\nModel 2: pack widgets up to max package weight of {0}"
            .format(packMax))
        self.initialize()
        self.r = random.Random()
        self.r.seed(initialSeed)
        wBuffer = Store(name="WidgetBuffer", capacity=500, sim=self)
        for i in range(nProd):
            wm = WidgetMakerW(name="WidgetMaker{0}".format(i), sim=self)
            self.activate(wm, wm.make(wBuffer))
        wp = WidgetPackerW(name="WidgetPacker", sim=self)
        self.activate(wp, wp.pack(wBuffer))
        self.simulate(until=simTime)


# Experiment ------------------------------
WidgetPackingModel2().run()
```

```
WidgetPacking
Model 1: pack 6 widgets per package
6.925710569651607: package completed
12.932859800562765: package completed
19.296609507200795: package completed
25.827518016011588: package completed
32.28961167417064: package completed
38.27653780313223: package completed
43.516005487393166: package completed
49.84136801349117: package completed
56.02183282676389: package completed
61.22427700160065: package completed
66.78084592723073: package completed
73.45267491487972: package completed
78.52559623726368: package completed
84.36238918070774: package completed
91.00334807967428: package completed
96.25112005144973: package completed

Model 2: pack widgets up to max package weight of 120
1.8535993255800536: package completed. Weight=  20.49 kg
2.9447148714144333: package completed. Weight=  43.29 kg
3.443679099583584: package completed. Weight=  60.73 kg
5.111979202993875: package completed. Weight=  81.36 kg
5.495036360384358: package completed. Weight= 103.32 kg
7.53958534514355: package completed. Weight=  43.79 kg
9.454524267565933: package completed. Weight=  62.32 kg
9.704524267565933: package completed. Weight=  82.07 kg
11.223206345291937: package completed. Weight= 103.55 kg
13.534262617168128: package completed. Weight=  41.81 kg
13.930945836966112: package completed. Weight=  62.71 kg
15.109021173114078: package completed. Weight=  82.73 kg
15.9312940569179: package completed. Weight= 103.41 kg
18.483473187078456: package completed. Weight=  37.07 kg
19.489762737880447: package completed. Weight=  56.79 kg
20.019035270021426: package completed. Weight=  77.23 kg
21.516971139857024: package completed. Weight=  96.93 kg
21.922330284736397: package completed. Weight= 116.85 kg
24.14512998963336: package completed. Weight=  37.89 kg
24.39877824367329: package completed. Weight=  55.50 kg
26.67484997458752: package completed. Weight=  78.46 kg
```

```
26.92484997458752: package completed. Weight= 100.35 kg
28.1664758950799: package completed. Weight= 117.54 kg
29.69269527817336: package completed. Weight=  40.05 kg
31.039995345892564: package completed. Weight=  60.53 kg
31.416541860477725: package completed. Weight=  77.63 kg
32.41760810512986: package completed. Weight= 100.53 kg
32.95461868723966: package completed. Weight= 118.77 kg
35.03241015535913: package completed. Weight=  40.09 kg
37.13500099371781: package completed. Weight=  59.57 kg
37.54993412593156: package completed. Weight=  76.58 kg
39.30652910593313: package completed. Weight=  95.59 kg
39.55652910593313: package completed. Weight= 116.49 kg
42.0420842938092: package completed. Weight=  42.24 kg
42.2920842938092: package completed. Weight=  63.58 kg
43.65741443896232: package completed. Weight=  85.90 kg
43.90741443896232: package completed. Weight= 106.78 kg
46.0951882462178: package completed. Weight=  37.20 kg
47.438300452780354: package completed. Weight=  58.47 kg
47.688300452780354: package completed. Weight=  79.33 kg
49.40690976731008: package completed. Weight= 100.51 kg
50.09589376227701: package completed. Weight= 117.80 kg
52.015927666123616: package completed. Weight=  37.47 kg
53.308294328826435: package completed. Weight=  58.78 kg
54.03538112053091: package completed. Weight=  81.10 kg
55.13265226734849: package completed. Weight=  98.94 kg
56.54061780546726: package completed. Weight=  39.67 kg
58.42284478038792: package completed. Weight=  61.78 kg
58.67284478038792: package completed. Weight=  80.36 kg
60.363861577200076: package completed. Weight= 102.50 kg
62.30054431149251: package completed. Weight=  38.39 kg
62.72667501529395: package completed. Weight=  58.21 kg
64.04176585171321: package completed. Weight=  75.31 kg
64.78871826102055: package completed. Weight=  94.49 kg
65.5200550350699: package completed. Weight= 113.54 kg
67.70604486309713: package completed. Weight=  41.53 kg
69.2327615315688: package completed. Weight=  60.57 kg
69.64556242167215: package completed. Weight=  82.52 kg
70.56237737140228: package completed. Weight= 104.26 kg
72.1477929466095: package completed. Weight=  41.62 kg
72.98432452690808: package completed. Weight=  63.88 kg
73.5826902990221: package completed. Weight=  81.87 kg
75.05025424304776: package completed. Weight= 101.04 kg
77.26048662562239: package completed. Weight=  40.26 kg
77.71710036932708: package completed. Weight=  61.69 kg
79.75353580124: package completed. Weight=  80.72 kg
80.12811805319318: package completed. Weight= 100.29 kg
82.4142311699317: package completed. Weight=  39.35 kg
83.68305885096552: package completed. Weight=  57.13 kg
83.93305885096552: package completed. Weight=  74.55 kg
85.85481524806448: package completed. Weight=  93.29 kg
86.13093377784419: package completed. Weight= 115.05 kg
87.86043544925184: package completed. Weight=  41.13 kg
89.46190021907331: package completed. Weight=  63.30 kg
90.40626417525424: package completed. Weight=  82.59 kg
91.19235409838856: package completed. Weight= 105.00 kg
92.48486291641056: package completed. Weight=  37.14 kg
93.79148753229941: package completed. Weight=  56.83 kg
94.19510234363693: package completed. Weight=  76.57 kg
```

```
96.09493981469757: package completed. Weight=  94.85 kg
96.82617211079959: package completed. Weight= 114.58 kg
98.82720299189798: package completed. Weight=  39.12 kg
99.94853712506526: package completed. Weight=  58.33 kg
```

GUI Input

The GUI examples do not run under Python 3.x, as only the core SimPy libraries were ported.

Fireworks using SimGUI: GUIdemo.py, GUIdemo_OO.py

A firework show. This is a very basic model, demonstrating the ease of interfacing to SimGUI.

```
__doc__ = """ GUIdemo.py
This is a very basic model, demonstrating the ease
of interfacing to SimGUI.
"""
from SimPy.Simulation import *
from random import *
from SimPy.SimGUI import *

# Model components ------------------------


class Launcher(Process):
    nrLaunched = 0

    def launch(self):
        while True:
            gui.writeConsole("Launch at {0:.1f}".format(now()))
            Launcher.nrLaunched += 1
            gui.launchmonitor.observe(Launcher.nrLaunched)
            yield hold, self, uniform(1, gui.params.maxFlightTime)
            gui.writeConsole("Boom!!! Aaaah!! at {0:.1f}".format(now()))


def model():
    gui.launchmonitor = Monitor(name="Rocket counter",
                                ylab="nr launched", tlab="time")
    initialize()
    Launcher.nrLaunched = 0
    for i in range(gui.params.nrLaunchers):
        lau = Launcher()
        activate(lau, lau.launch())
    simulate(until=gui.params.duration)
    gui.noRunYet = False
    gui.writeStatusLine("{0} rockets launched in {1:.1f} minutes"
                        .format(Launcher.nrLaunched, now()))


class MyGUI(SimGUI):
    def __init__(self, win, **par):
        SimGUI.__init__(self, win, **par)
        self.run.add_command(label="Start fireworks",
                             command=model, underline=0)
```

```
        self.params = Parameters(duration=duration,
                                 maxFlightTime=maxFlightTime,
                                 nrLaunchers=nrLaunchers)

# Experiment data ------------------------------------


duration = 2000
maxFlightTime = 11.7
nrLaunchers = 3

# Model/Experiment/Display  --------------------------

root = Tk()
gui = MyGUI(root, title="RocketGUI", doc=__doc__, consoleHeight=40)
gui.mainloop()
```

Here is the OO version:

```
__doc__ = """ GUIdemo_OO.py
This is a very basic model, demonstrating the ease
of interfacing to SimGUI.
"""
from SimPy.Simulation import *
from random import *
from SimPy.SimGUI import *

# Model components ------------------------------------


class Launcher(Process):
    nrLaunched = 0

    def launch(self):
        while True:
            gui.writeConsole("Launch at {0:.1f}".format(self.sim.now()))
            Launcher.nrLaunched += 1
            gui.launchmonitor.observe(Launcher.nrLaunched)
            yield hold, self, uniform(1, gui.params.maxFlightTime)
            gui.writeConsole(
                "Boom!!! Aaaah!! at {0:.1f}".format(self.sim.now()))

# Model ----------------------------------------------


class GUIdemoModel(Simulation):
    def run(self):
        self.initialize()
        gui.launchmonitor = Monitor(name="Rocket counter",
                                    ylab="nr launched", tlab="time", sim=self)
        Launcher.nrLaunched = 0
        for i in range(gui.params.nrLaunchers):
            lau = Launcher(sim=self)
            self.activate(lau, lau.launch())
        self.simulate(until=gui.params.duration)
        gui.noRunYet = False
        gui.writeStatusLine("{0} rockets launched in {1:.1f} minutes"
                            .format(Launcher.nrLaunched, self.now()))
```

```
# Model GUI ------------------------------------------


class MyGUI(SimGUI):
    def __init__(self, win, **par):
        SimGUI.__init__(self, win, **par)
        self.run.add_command(label="Start fireworks",
                             command=GUIdemoModel().run, underline=0)
        self.params = Parameters(duration=duration,
                                 maxFlightTime=maxFlightTime,
                                 nrLaunchers=nrLaunchers)

# Experiment data ------------------------------------


duration = 2000
maxFlightTime = 11.7
nrLaunchers = 3

# Experiment/Display  --------------------------------

root = Tk()
gui = MyGUI(root, title="RocketGUI", doc=__doc__, consoleHeight=40)
gui.mainloop()
```

Bank Customers using SimGUI: bank11GUI.py, bank11GUI_OO.py

Simulation with customers arriving at random to a bank with two counters. This is a modification of the bank11 simulation using SimGUI to run the simulation.

```
__doc__ = """ bank11.py: Simulate customers arriving
    at random, using a Source, requesting service
    from two counters each with their own queue
    random servicetime.
    Uses a Monitor object to record waiting times
"""
from SimPy.Simulation import *
# Lmona
from random import Random
from SimPy.SimGUI import *


class Source(Process):
    """ Source generates customers randomly"""

    def __init__(self, seed=333):
        Process.__init__(self)
        self.SEED = seed

    def generate(self, number, interval):
        rv = Random(self.SEED)
        for i in range(number):
            c = Customer(name="Customer{0:02d}".format(i))
            activate(c, c.visit(timeInBank=12.0))
            t = rv.expovariate(1.0 / interval)
```

```
            yield hold, self, t


def NoInSystem(R):
    """ The number of customers in the resource R
    in waitQ and active Q"""
    return (len(R.waitQ) + len(R.activeQ))


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def __init__(self, name):
        Process.__init__(self)
        self.name = name

    def visit(self, timeInBank=0):
        arrive = now()
        Qlength = [NoInSystem(counter[i]) for i in range(Nc)]
        if gui.params.trace:
            gui.writeConsole("{0:7.4f} {1}: Here I am. Queues are: {2}".format(
                now(), self.name, Qlength))
        for i in range(Nc):
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                join = i
                break
        yield request, self, counter[join]
        wait = now() - arrive
        waitMonitor.observe(wait, t=now())
        if gui.params.trace:
            gui.writeConsole("{0:7.4f} {1}: Waited {2:6.3f}".format(
                now(), self.name, wait))
        tib = counterRV.expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter[join]
        if gui.params.trace:
            gui.writeConsole(
                "{0:7.4f} {1}: Finished    ".format(now(), self.name))


def model(counterseed=3939393):
    global Nc, counter, counterRV, waitMonitor
    Nc = 2
    counter = [Resource(name="Clerk0"), Resource(name="Clerk1")]
    counterRV = Random(counterseed)
    gui.mon1 = waitMonitor = Monitor("Customer waiting times")
    waitMonitor.tlab = "arrival time"
    waitMonitor.ylab = "waiting time"
    initialize()
    source = Source(seed=gui.params.sourceseed)
    activate(source, source.generate(
        gui.params.numberCustomers, gui.params.interval), 0.0)
    result = simulate(until=gui.params.endtime)
    gui.writeStatusLine(text="Time at simulation end: {0:.1f}"
                        " -- Customers still waiting: {1}"
                        .format(now(),
                                len(counter[0].waitQ) +
                                len(counter[1].waitQ)))
```

```
def statistics():
    if gui.noRunYet:
        showwarning(title='Model warning',
                    message="Run simulation first -- no data available.")
        return
    gui.writeConsole(text="\nRun parameters: {0}".format(gui.params))
    gui.writeConsole(text="Average wait for {0:4d} customers was {1:6.2f}"
                     .format(waitMonitor.count(), waitMonitor.mean()))


def run():
    model(gui.params.counterseed)
    gui.noRunYet = False


def showAuthors():
    gui.showTextBox(text="Tony Vignaux\nKlaus Muller",
                    title="Author information")


class MyGUI(SimGUI):
    def __init__(self, win, **p):
        SimGUI.__init__(self, win, **p)
        self.help.add_command(label="Author(s)",
                              command=showAuthors, underline=0)
        self.view.add_command(label="Statistics",
                              command=statistics, underline=0)
        self.run.add_command(label="Run bank11 model",
                             command=run, underline=0)

        self.params = Parameters(
            endtime=2000,
            sourceseed=1133,
            counterseed=3939393,
            numberCustomers=50,
            interval=10.0,
            trace=0)


root = Tk()
gui = MyGUI(root, title="SimPy GUI example", doc=__doc__, consoleHeight=40)
gui.mainloop()
```

Here is the OO version:

```
__doc__ = """ bank11_OO.py: Simulate customers arriving
    at random, using a Source, requesting service
    from two counters each with their own queue
    random servicetime.
    Uses a Monitor object to record waiting times
"""
from SimPy.Simulation import *
from random import Random
from SimPy.SimGUI import *
```

```
class Source(Process):
    """ Source generates customers randomly"""

    def __init__(self, sim, seed=333):
        Process.__init__(self, sim=sim)
        self.SEED = seed

    def generate(self, number, interval):
        rv = Random(self.SEED)
        for i in range(number):
            c = Customer(name="Customer{0:02d}".format(i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            t = rv.expovariate(1.0 / interval)
            yield hold, self, t


def NoInSystem(R):
    """ The number of customers in the resource R
    in waitQ and active Q"""
    return (len(R.waitQ) + len(R.activeQ))


class Customer(Process):
    """ Customer arrives, is served and leaves """
    # def __init__(self,name,sim):
    # Process.__init__(self,name=name,sim=sim)

    def visit(self, timeInBank=0):
        arrive = self.sim.now()
        Qlength = [NoInSystem(self.sim.counter[i]) for i in range(self.sim.Nc)]
        if gui.params.trace:
            gui.writeConsole("{0:7.4f} {1}: Here I am. Queues are: {2}".format(
                self.sim.now(), self.name, Qlength))
        for i in range(self.sim.Nc):
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                join = i
                break
        yield request, self, self.sim.counter[join]
        wait = self.sim.now() - arrive
        self.sim.waitMonitor.observe(wait, t=self.sim.now())
        if gui.params.trace:
            gui.writeConsole("{0:7.4f} {1}: Waited {2:6.3f}".format(
                self.sim.now(), self.name, wait))
        tib = self.sim.counterRV.expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.sim.counter[join]
        if gui.params.trace:
            gui.writeConsole("{0:7.4f} {1}: Finished    ".format(
                self.sim.now(), self.name))


class CounterModel(Simulation):
    def run(self, counterseed=3939393):
        self.initialize()
        self.Nc = 2
        self.counter = [Resource(name="Clerk0", sim=self),
                        Resource(name="Clerk1", sim=self)]
        self.counterRV = Random(counterseed)
```

```
        gui.mon1 = self.waitMonitor = Monitor(
            "Customer waiting times", sim=self)
        self.waitMonitor.tlab = "arrival time"
        self.waitMonitor.ylab = "waiting time"
        source = Source(seed=gui.params.sourceseed, sim=self)
        self.activate(source, source.generate(
            gui.params.numberCustomers, gui.params.interval), 0.0)
        result = self.simulate(until=gui.params.endtime)
        gui.writeStatusLine(text="Time at simulation end: {0:.1f}"
                            " -- Customers still waiting: {1}"
                            .format(self.now(),
                                    len(self.counter[0].waitQ) +
                                    len(self.counter[1].waitQ)))


def statistics():
    if gui.noRunYet:
        showwarning(title='Model warning',
                    message="Run simulation first -- no data available.")
        return
    gui.writeConsole(text="\nRun parameters: {0}".format(gui.params))
    gui.writeConsole(text="Average wait for {0:4d} customers was {1:6.2f}"
                     .format(gui.mon1.count(), gui.mon1.mean()))


def run():
    CounterModel().run(gui.params.counterseed)
    gui.noRunYet = False


def showAuthors():
    gui.showTextBox(text="Tony Vignaux\nKlaus Muller",
                    title="Author information")


class MyGUI(SimGUI):
    def __init__(self, win, **p):
        SimGUI.__init__(self, win, **p)
        self.help.add_command(label="Author(s)",
                              command=showAuthors, underline=0)
        self.view.add_command(label="Statistics",
                              command=statistics, underline=0)
        self.run.add_command(label="Run bank11 model",
                             command=run, underline=0)

        self.params = Parameters(
            endtime=2000,
            sourceseed=1133,
            counterseed=3939393,
            numberCustomers=50,
            interval=10.0,
            trace=0)


root = Tk()
gui = MyGUI(root, title="SimPy GUI example", doc=__doc__, consoleHeight=40)
gui.mainloop()
```

Bank Customers using SimulationStep: SimGUIStep.py

(broken for python 2.x global name 'simulateStep' is not defined)

Another modification of the bank11 simulation this time showing the ability to step between events.

```
from SimPy.SimGUI import *
from SimPy.SimulationStep import *
from random import Random

__version__ = '$Revision$ $Date$ kgm'

if __name__ == '__main__':

    class Source(Process):
        """ Source generates customers randomly"""

        def __init__(self, seed=333):
            Process.__init__(self)
            self.SEED = seed

        def generate(self, number, interval):
            rv = Random(self.SEED)
            for i in range(number):
                c = Customer(name="Customer{0:02d}".format(i,))
                activate(c, c.visit(timeInBank=12.0))
                t = rv.expovariate(1.0 / interval)
                yield hold, self, t

    def NoInSystem(R):
        """ The number of customers in the resource R
        in waitQ and active Q"""
        return (len(R.waitQ) + len(R.activeQ))

    class Customer(Process):
        """ Customer arrives, is served and leaves """

        def __init__(self, name):
            Process.__init__(self)
            self.name = name

        def visit(self, timeInBank=0):
            arrive = now()
            Qlength = [NoInSystem(counter[i]) for i in range(Nc)]
            # print("{0:7.4f} {1}: Here I am. {2}    ".format(now(),
            #                                                  self.name,
            #                                                  Qlength))
            for i in range(Nc):
                if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                    join = i
                    break
            yield request, self, counter[join]
            wait = now() - arrive
            waitMonitor.observe(wait, t=now())  # Lmond
            # print("{0:7.4f} {1}: Waited {2:6.3f}".format(now(),
            #                                              self.name,wait))
            tib = counterRV.expovariate(1.0 / timeInBank)
            yield hold, self, tib
```

```
            yield release, self, counter[join]
            serviceMonitor.observe(now() - arrive, t=now())
            if trace:
                gui.writeConsole("Customer leaves at {0:.1f}".format(now()))
            # print("{0:7.4f} {1}: Finished    ".format(now(),self.name))

    def showtime():
        gui.topconsole.config(text="time = {0}".format(now()))
        gui.root.update()

    def runStep():
        if gui.noRunYet:
            showwarning("SimPy warning", "Run 'Start run (stepping)' first")
            return
        showtime()
        a = simulateStep(until=gui.params.endtime)
        if a[1] == "notResumable":
            gui.writeConsole(text="Run ended. Status: {0}".format(a[0]))
        showtime()

    def runNoStep():
        showtime()
        for i in range(gui.params.nrRuns):
            simulate(until=gui.param.sendtime)
        showtime()
        gui.writeConsole("{0} simulation run(s) completed\n".format(i + 1))

    def contStep():
        return

    def model():
        global Nc, counter, counterRV, waitMonitor, serviceMonitor, trace
        global lastLeave, noRunYet, initialized
        counterRV = Random(gui.params.counterseed)
        sourceseed = gui.params.sourceseed
        nrRuns = gui.params.nrRuns
        lastLeave = 0
        gui.noRunYet = True
        for runNr in range(nrRuns):
            gui.noRunYet = False
            trace = gui.params.trace
            if trace:
                gui.writeConsole(text='\n** Run {0}'.format(runNr + 1))
            Nc = 2
            counter = [Resource(name="Clerk0"), Resource(name="Clerk1")]
            gui.waitMoni = waitMonitor = Monitor(name='Waiting Times')
            waitMonitor.xlab = 'Time'
            waitMonitor.ylab = 'Customer waiting time'
            gui.serviceMoni = serviceMonitor = Monitor(name='Service Times')
            serviceMonitor.xlab = 'Time'
            serviceMonitor.ylab = 'Total service time = wait+service'
            initialize()
            source = Source(seed=sourceseed)
            activate(source, source.generate(
                gui.params.numberCustomers, gui.params.interval), 0.0)
            simulate(showtime, until=gui.params.endtime)
            showtime()
            lastLeave += now()
```

```
        gui.writeConsole("{0} simulation run(s) completed\n".format(nrRuns))
        gui.writeConsole("Parameters:\n{0}".format(gui.params))

    def modelstep():
        global Nc, counter, counterRV, waitMonitor, serviceMonitor, trace
        global lastLeave, noRunYet
        counterRV = Random(gui.params.counterseed)
        sourceseed = gui.params.sourceseed
        nrRuns = gui.params.nrRuns
        lastLeave = 0
        gui.noRunYet = True
        trace = gui.params.trace
        if trace:
            gui.writeConsole(text='\n** Run {0}'.format(runNr + 1))
        Nc = 2
        counter = [Resource(name="Clerk0"), Resource(name="Clerk1")]
        gui.waitMoni = waitMonitor = Monitor(name='Waiting Times')
        waitMonitor.xlab = 'Time'
        waitMonitor.ylab = 'Customer waiting time'
        gui.serviceMoni = serviceMonitor = Monitor(name='Service Times')
        serviceMonitor.xlab = 'Time'
        serviceMonitor.ylab = 'Total service time = wait+service'
        initialize()
        source = Source(seed=sourceseed)
        activate(source, source.generate(
            gui.params.numberCustomers, gui.params.interval), 0.0)
        simulateStep(until=gui.params.endtime)
        gui.noRunYet = False

    def statistics():
        if gui.noRunYet:
            showwarning(title='SimPy warning',
                        message="Run simulation first -- no data available.")
            return
        aver = lastLeave / gui.params.nrRuns
        gui.writeConsole(text="Average time for {0} customers to get through "
                              "bank: {1:.1f}\n({2} runs)\n"
                              .format(gui.params.numberCustomers, aver,
                                      gui.params.nrRuns))

    __doc__ = """
Modified bank11.py (from Bank Tutorial) with GUI.

Model: Simulate customers arriving
at random, using a Source, requesting service
from two counters each with their own queue
random servicetime.

Uses Monitor objects to record waiting times
and total service times."""

    def showAuthors():
        gui.showTextBox(text="Tony Vignaux\nKlaus Muller",
                        title="Author information")

    class MyGUI(SimGUI):
        def __init__(self, win, **p):
            SimGUI.__init__(self, win, **p)
```

```
            self.help.add_command(label="Author(s)",
                                  command=showAuthors, underline=0)
            self.view.add_command(label="Statistics",
                                  command=statistics, underline=0)
            self.run.add_command(label="Start run (event stepping)",
                                 command=modelstep, underline=0)
            self.run.add_command(label="Next event",
                                 command=runStep, underline=0)
            self.run.add_command(label="Complete run (no stepping)",
                                 command=model, underline=0)

    root = Tk()
    gui = MyGUI(root, title="SimPy GUI example", doc=__doc__)
    gui.params = Parameters(endtime=2000,
                            sourceseed=1133,
                            counterseed=3939393,
                            numberCustomers=50,
                            interval=10.0,
                            trace=0,
                            nrRuns=1)
    gui.mainloop()
```

Plot

Patisserie Francaise bakery: bakery.py, bakery_OO.py

The Patisserie Francaise bakery has three ovens baking their renowned baguettes for retail and restaurant customers. They start baking one hour before the shop opens and stop at closing time.

They bake batches of 40 breads at a time, taking 25..30 minutes (uniformly distributed) per batch. Retail customers arrive at a rate of 40 per hour (exponentially distributed). They buy 1, 2 or 3 baguettes with equal probability. Restaurant buyers arrive at a rate of 4 per hour (exponentially dist.). They buy 20,40 or 60 baguettes with equal probability.

The simulation answers the following questions:

a) What is the mean waiting time for retail and restaurant buyers?

2. What is their maximum waiting time?
3. What percentage of customer has to wait longer than 15 minutes?

SimPy.SimPlot is used to graph the number of baguettes over time. (KGM)

```
"""bakery.py
Scenario:
The Patisserie Francaise bakery has three ovens baking their renowned
baguettes for retail and restaurant customers. They start baking one
hour before the shop opens and stop at closing time.
They bake batches of 40 breads at a time,
taking 25..30 minutes (uniformly distributed) per batch. Retail customers
arrive at a rate of 40 per hour (exponentially distributed). They buy
1, 2 or 3 baguettes with equal probability. Restaurant buyers arrive
at a rate of 4 per hour (exponentially dist.). They buy 20,40 or 60
baguettes with equal probability.
Simulate this operation for 100 days of 8 hours shop opening time.
a) What is the mean waiting time for retail and restaurant buyers?
b) What is their maximum waiting time?
b) What percentage of customer has to wait longer than 15 minutes??
```

```
c) Plot the number of baguettes over time for an arbitrary day.
   (use PLOTTING=True to do this)

"""
from SimPy.Simulation import *
from SimPy.SimPlot import *
import random
# Model components


class Bakery:
    def __init__(self, nrOvens, toMonitor):
        self.stock = Level(name="baguette stock", monitored=toMonitor)
        for i in range(nrOvens):
            ov = Oven()
            activate(ov, ov.bake(capacity=batchsize, bakery=self))


class Oven(Process):
    def bake(self, capacity, bakery):
        while now() + tBakeMax < tEndBake:
            yield hold, self, r.uniform(tBakeMin, tBakeMax)
            yield put, self, bakery.stock, capacity


class Customer(Process):
    def buyBaguette(self, cusType, bakery):
        tIn = now()
        yield get, self, bakery.stock, r.choice(buy[cusType])
        waits[cusType].append(now() - tIn)


class CustomerGenerator(Process):
    def generate(self, cusType, bakery):
        while True:
            yield hold, self, r.expovariate(1.0 / tArrivals[cusType])
            if now() < (tShopOpen + tBeforeOpen):
                c = Customer(cusType)
                activate(c, c.buyBaguette(cusType, bakery=bakery))
# Model


def model():
    toMonitor = False
    initialize()
    if day == (nrDays - 1):
        toMoni = True
    else:
        toMoni = False
    b = Bakery(nrOvens=nrOvens, toMonitor=toMoni)
    for cType in ["retail", "restaurant"]:
        cg = CustomerGenerator()
        activate(cg, cg.generate(cusType=cType, bakery=b), delay=tBeforeOpen)
    simulate(until=tBeforeOpen + tShopOpen)
    return b


# Experiment data
```

```
nrOvens = 3
batchsize = 40  # nr baguettes
tBakeMin = 25 / 60.
tBakeMax = 30 / 60.  # hours
tArrivals = {"retail": 1.0 / 40, "restaurant": 1.0 / 4}  # hours
buy = {"retail": [1, 2, 3], "restaurant": [20, 40, 60]}  # nr baguettes
tShopOpen = 8
tBeforeOpen = 1
tEndBake = tBeforeOpen + tShopOpen  # hours
nrDays = 100
r = random.Random(12371)
PLOTTING = True
# Experiment
waits = {}
waits["retail"] = []
waits["restaurant"] = []
for day in range(nrDays):
    bakery = model()
# Analysis/output
print("bakery")
for cType in ["retail", "restaurant"]:
    print("Average wait for {0} customers: {1:4.2f} hours".format(
        cType, (1.0 * sum(waits[cType])) / len(waits[cType])))
    print("Longest wait for {0} customers: {1:4.1f} hours".format(
        cType, max(waits[cType])))
    nrLong = len([1 for x in waits[cType] if x > 0.25])
    nrCust = len(waits[cType])
    print("Percentage of {0} customers having to wait for more than"
          " 0.25 hours: {1}".format(cType, 100 * nrLong / nrCust))


if PLOTTING:
    plt = SimPlot()
    plt.plotStep(bakery.stock.bufferMon,
                 title="Number of baguettes in stock during arbitrary day",
                 color="blue")
    plt.mainloop()
```

Here is the OO version:

```
"""bakery_OO.py
Scenario:
The Patisserie Francaise bakery has three ovens baking their renowned
baguettes for retail and restaurant customers. They start baking one
hour before the shop opens and stop at closing time.
They bake batches of 40 breads at a time,
taking 25..30 minutes (uniformly distributed) per batch. Retail customers
arrive at a rate of 40 per hour (exponentially distributed). They buy
1, 2 or 3 baguettes with equal probability. Restaurant buyers arrive
at a rate of 4 per hour (exponentially dist.). They buy 20,40 or 60
baguettes with equal probability.
Simulate this operation for 100 days of 8 hours shop opening time.
a) What is the mean waiting time for retail and restaurant buyers?
b) What is their maximum waiting time?
b) What percentage of customer has to wait longer than 15 minutes??
c) Plot the number of baguettes over time for an arbitrary day.
   (use PLOTTING=True to do this)
```

```
"""
from SimPy.Simulation import *
from SimPy.SimPlot import *
import random
# Model components ------------------------


class Bakery:
    def __init__(self, nrOvens, toMonitor, sim):
        self.stock = Level(name="baguette stock", monitored=toMonitor, sim=sim)
        for i in range(nrOvens):
            ov = Oven(sim=sim)
            sim.activate(ov, ov.bake(capacity=batchsize, bakery=self))


class Oven(Process):
    def bake(self, capacity, bakery):
        while self.sim.now() + tBakeMax < tEndBake:
            yield hold, self, r.uniform(tBakeMin, tBakeMax)
            yield put, self, bakery.stock, capacity


class Customer(Process):
    def buyBaguette(self, cusType, bakery):
        tIn = self.sim.now()
        yield get, self, bakery.stock, r.choice(buy[cusType])
        waits[cusType].append(self.sim.now() - tIn)


class CustomerGenerator(Process):
    def generate(self, cusType, bakery):
        while True:
            yield hold, self, r.expovariate(1.0 / tArrivals[cusType])
            if self.sim.now() < (tShopOpen + tBeforeOpen):
                c = Customer(cusType, sim=self.sim)
                self.sim.activate(c, c.buyBaguette(cusType, bakery=bakery))
# Model -----------------------------------


class BakeryModel(Simulation):
    def run(self):
        # toMonitor=False
        self.initialize()
        toMoni = day == (nrDays - 1)
        b = Bakery(nrOvens=nrOvens, toMonitor=toMoni, sim=self)
        for cType in ["retail", "restaurant"]:
            cg = CustomerGenerator(sim=self)
            self.activate(cg, cg.generate(
                cusType=cType, bakery=b), delay=tBeforeOpen)
        self.simulate(until=tBeforeOpen + tShopOpen)
        return b


# Experiment data -------------------------
nrOvens = 3
batchsize = 40  # nr baguettes
tBakeMin = 25 / 60.
tBakeMax = 30 / 60.  # hours
```

```
tArrivals = {"retail": 1.0 / 40, "restaurant": 1.0 / 4}  # hours
buy = {"retail": [1, 2, 3], "restaurant": [20, 40, 60]}  # nr baguettes
tShopOpen = 8
tBeforeOpen = 1
tEndBake = tBeforeOpen + tShopOpen  # hours
nrDays = 100
r = random.Random(12371)
PLOTTING = True
# Experiment ------------------------------
waits = {}
waits["retail"] = []
waits["restaurant"] = []
bakMod = BakeryModel()
for day in range(nrDays):
    bakery = bakMod.run()
# Analysis/output -------------------------
print('bakery_OO')
for cType in ["retail", "restaurant"]:
    print("Average wait for {0} customers: {1:4.2f} hours"
          .format(cType, (1.0 * sum(waits[cType])) / len(waits[cType])))
    print("Longest wait for {0} customers: {1:4.1f} hours"
          .format(cType, max(waits[cType])))
    nrLong = len([1 for x in waits[cType] if x > 0.25])
    nrCust = len(waits[cType])
    print("Percentage of {0} customers having to wait for more than"
          " 0.25 hours: {1}".format(cType, 100 * nrLong / nrCust))


if PLOTTING:
    plt = SimPlot()
    plt.plotStep(bakery.stock.bufferMon,
                 title="Number of baguettes in stock during arbitrary day",
                 color="blue")
    plt.mainloop()
```

Bank Customers Demos SimPlot: bank11Plot.py

A modification of the bank11 simulation with graphical output. It plots service and waiting times.

```
""" bank11.py: Simulate customers arriving
    at random, using a Source, requesting service
    from two counters each with their own queue
    random servicetime.
    Uses a Monitor object to record waiting times
"""
from SimPy.Simulation import *
from SimPy.SimPlot import *
from random import Random


class Bank11(SimPlot):
    def __init__(self, **p):
        SimPlot.__init__(self, **p)

    def NoInSystem(self, R):
        """ The number of customers in the resource R
```

```
        in waitQ and active Q"""
        return (len(R.waitQ) + len(R.activeQ))

    def model(self):
        self.counterRV = Random(self.params["counterseed"])
        self.sourceseed = self.params["sourceseed"]
        nrRuns = self.params["nrRuns"]
        self.lastLeave = 0
        self.noRunYet = True
        for runNr in range(nrRuns):
            self.noRunYet = False
            self.Nc = 2
            self.counter = [Resource(name="Clerk0", monitored=False),
                            Resource(name="Clerk1", monitored=False)]
            self.waitMonitor = Monitor(name='Waiting Times')
            self.waitMonitor.xlab = 'Time'
            self.waitMonitor.ylab = 'Waiting time'
            self.serviceMonitor = Monitor(name='Service Times')
            self.serviceMonitor.xlab = 'Time'
            self.serviceMonitor.ylab = 'wait+service'
            initialize()
            source = Source(self, seed=self.sourceseed * 1000)
            activate(source, source.generate(self.params["numberCustomers"],
                                             self.params["interval"]), 0.0)
            simulate(until=self.params['endtime'])
            self.lastLeave += now()
        print("{0} run(s) completed".format(nrRuns))
        print("Parameters:\n{0}".format(self.params))


class Source(Process):
    """ Source generates customers randomly"""

    def __init__(self, modInst, seed=333):
        Process.__init__(self)
        self.modInst = modInst
        self.SEED = seed

    def generate(self, number, interval):
        rv = Random(self.SEED)
        for i in range(number):
            c = Customer(self.modInst, name="Customer{0:02d}".format(i))
            activate(c, c.visit(timeInBank=12.0))
            t = rv.expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def __init__(self, modInst, **p):
        Process.__init__(self, **p)
        self.modInst = modInst

    def visit(self, timeInBank=0):
        arrive = now()
        Qlength = [self.modInst.NoInSystem(self.modInst.counter[i])
                   for i in range(self.modInst.Nc)]
```

```
        for i in range(self.modInst.Nc):
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                join = i
                break
        yield request, self, self.modInst.counter[join]
        wait = now() - arrive
        self.modInst.waitMonitor.observe(wait, t=now())
        # print("{0:7.4f} {1}: Waited {2:6.3f}".format(now(),self.name,wait))
        tib = self.modInst.counterRV.expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.modInst.counter[join]
        self.modInst.serviceMonitor.observe(now() - arrive, t=now())


root = Tk()
plt = Bank11()
plt.params = {"endtime": 2000,
              "sourceseed": 1133,
              "counterseed": 3939393,
              "numberCustomers": 50,
              "interval": 10.0,
              "trace": 0,
              "nrRuns": 1}
plt.model()
plt.plotLine(plt.waitMonitor, color='blue', width=2)
plt.plotLine(plt.serviceMonitor, color='red', width=2)
root.mainloop()
```

Debugger

Stepping thru Simpulation Events: SimpleDebugger.py

A utility module for stepping through the events of a simulation under user control, using SimulationTrace.

```
"""
A utility module for stepping through the events of a simulation
under user control.

REQUIRES SimPy 2.1
"""
from SimPy.SimulationTrace import *

from sys import version_info  # Needed to determine if Python 2 or 3
py_ver = version_info[0]


def stepper():
    evlist = Globals.sim._timestamps
    while True:
        if not evlist:
            print("No more events.")
            break
        tEvt = evlist[0][0]
        who = evlist[0][2]
        while evlist[0][3]:  # skip cancelled event notices
            step()
```

```
        print("\nTime now: {0}, next event at: {1} for process: {2} ".format(
            now(), tEvt, who.name))  # peekAll()[0],peekAll()[1].name)

        if(py_ver > 2):  # python3
            cmd = input(
                "'s' next event,'r' run to end,'e' to end run, "
                "<time> skip to event at <time>, 'l' show eventlist, "
                "'p<name>' skip to event for <name>: ")
        else:  # python2
            cmd = raw_input(
                "'s' next event,'r' run to end,'e' to end run, "
                "<time> skip to event at <time>, 'l' show eventlist, "
                "'p<name>' skip to event for <name>: ")
        try:
            nexttime = float(cmd)
            while peek() < nexttime:
                step()
        except:
            if cmd == 's':
                step()
            elif cmd == 'r':
                break
            elif cmd == 'e':
                stopSimulation()
                break
            elif cmd == 'l':
                print("Events scheduled: \n{0}".format(allEventNotices()))
            elif cmd[0] == 'p':
                while evlist and evlist[0][2].name != cmd[1:]:
                    step()
            else:
                print("{0} not a valid command".format(cmd))


if __name__ == "__main__":
    import random as r
    r.seed(1234567)

    class Test(Process):
        def run(self):
            while now() < until:
                yield hold, self, r.uniform(1, 10)

    class Waiter(Process):
        def run(self, evt):
            def gt30():
                return now() > 30

            yield waituntil, self, gt30
            print("now() is past 30")
            stopSimulation()

    until = 100
    initialize()
    evt = SimEvent()
    t = Test("Test1")
    activate(t, t.run())
    t2 = Test("Test2")
```

```
    activate(t2, t2.run())
    w = Waiter("Waiter")
    activate(w, w.run(evt=evt))
    stepper()
```

Here is the OO version:

```
"""
A utility module for stepping through the events of a simulation
under user control.

REQUIRES SimPy 2.1
"""
from SimPy.SimulationTrace import *

from sys import version_info  # Needed to determine if using Python 2 or 3
py_ver = version_info[0]


def stepper(whichsim):
    evlist = whichsim._timestamps
    while True:
        if not evlist:
            print("No more events.")
            break
        tEvt = evlist[0][0]
        who = evlist[0][2]
        while evlist[0][3]:  # skip cancelled event notices
            whichsim.step()
        print("\nTime now: {0}, next event at: {1} for process: {2} ".format(
            whichsim.now(), tEvt, who.name))

        if(py_ver > 2):  # Python 3
            cmd = input(
                "'s' next event,'r' run to end,'e' to end run, "
                "<time> skip to event at <time>, 'l' show eventlist, "
                "'p<name>' skip to event for <name>: ")
        else:  # Python 2
            cmd = raw_input(
                "'s' next event,'r' run to end,'e' to end run, "
                "<time> skip to event at <time>, 'l' show eventlist, "
                "'p<name>' skip to event for <name>: ")
        try:
            nexttime = float(cmd)
            while whichsim.peek() < nexttime:
                whichsim.step()
        except:
            if cmd == 's':
                whichsim.step()
            elif cmd == 'r':
                break
            elif cmd == 'e':
                stopSimulation()
                break
            elif cmd == 'l':
                print("Events scheduled: \n{0}".format(
                    whichsim.allEventNotices()))
            elif cmd[0] == 'p':
```

```
                while evlist and evlist[0][2].name != cmd[1:]:
                    whichsim.step()
            else:
                print("{0} not a valid command".format(cmd))


if __name__ == "__main__":
    import random as r
    r.seed(1234567)

    class Test(Process):
        def run(self):
            while self.sim.now() < until:
                yield hold, self, r.uniform(1, 10)

    class Waiter(Process):
        def run(self, evt):
            def gt30():
                return self.sim.now() > 30

            yield waituntil, self, gt30
            print("now() is past 30")
            self.sim.stopSimulation()

    until = 100
    s = SimulationTrace()
    s.initialize()
    evt = SimEvent(sim=s)
    t = Test("Test1", sim=s)
    s.activate(t, t.run())
    t2 = Test("Test2", sim=s)
    s.activate(t2, t2.run())
    w = Waiter("Waiter", sim=s)
    s.activate(w, w.run(evt=evt))
    stepper(whichsim=s)
```

Authors

- Tony Vignaux <Vignaux@users.sourceforge.net>,
- Klaus Muller <Muller@users.sourceforge.net>
- Karen Turner (updated 2012)
- Steven Kennedy (updated for Python 3 2012)

Created 2002-December

Date 2012-April

2.9 Short Manual for SimPy Classic

SimPy Classic is a free, open-source discrete-event simulation system written in Python. It provides a number of tools for programmers writing simulation programs. This document is a description of basic techniques of SimPy. It describes a subset of SimPy's capabilities - sufficient, we think, to develop standard simulations. You may also find The Bank tutorial included in SimPy's distribution helpful in the early stages. The full SimPy Classic Manual, included in the distribution, is more detailed.

You need to write Python code to develop a SimPy Classic model. In particular, you will have to define and use classes and their objects. Python is free and open-source and is available on most platforms. You can find out more about it and download it from the Python web-site where there is full documentation and tutorials. SimPy requires Python version 2.3 or later.

Contents:

2.9.1 Introduction to SimPy Classic

Authors G A Vignaux and Klaus Muller

Date Feb 24, 2018

Release 2.3.3

Python-Version 2.7 and later

Contents

Introduction

The active elements (or *entities*) of a SimPy model are objects of a class defined by the programmer (see *Processes*, section 3). Each entity has a standard method, a Process Execution Method (referred to by SimPy programmers as a PEM) which specifies its actions in detail. Each PEM runs in parallel with (and may interact with) the PEMs of other entities.

The activity of an entity may be delayed for fixed or random times, queued at resource facilities, and may be interrupted by or interact in different ways with other entities and components. For example in a gas station model, automobile entities (objects of an Automobile Class) may have to wait at the gas station for a pump to become available. On obtaining a pump it takes time to fill the tank. The pump is then released for the next automobile in the queue if there is one.

SimPy has three kinds of resource facilities (Resources, Levels, and Stores). Each type models a congestion point where entities queue while waiting to acquire or, in some cases, to deposit a resource.SimPy automatically handles the queueing.

- *Resources* have one or more identical resource units, each of which can be held by entities. Extending the example above, the gas station might be modelled as a Resource with its pumps as resource units. When a car requests a pump the gas station resource automatically queues it until a pump becomes available (perhaps immediately). The car holds the pump until it finishes refuelling and then releases it for use by the next car.

- `Levels` (not treated here) model the supply and consumption of a homogeneous undifferentiated "material". The Level holds an amount that is fully described by a non-negative number which can be increased or decreased by entities. For example, a gas station stores gas in large storage tanks. The tanks can be filled by tankers and emptied by cars refuelling. In contrast to the operation of a Resource, a car need not return the gas to the gas station.
- `Stores` (not treated here) model the production and consumption of distinguishable items. A Store holds a list of items. Entities can insert or remove items from the list and these can be of any type. They can even be SimPy process objects. For example, the gas station holds spares of different types. A car might request a set of spares from the Store. The store is replenished by deliveries from a warehouse.

SimPy also supplies Monitors and Tallys to record simulation events. *Monitors* are used to compile summary statistics such as waiting times and queue lengths. These statistics includes simple averages and variances, time-weighted averages, or histograms. In particular, data can be gathered on the queues associated with Resources, Levels and Stores. For example we may collect data on the average number of cars waiting at the gas station and the distribution of their waiting times. Monitors preserve complete time-series records that may later be used for more advanced post-simulation analyses. A simpler version of a Monitor is a Tally which provides most of the statistical facilities but does not retain a complete time-series record. See the SimPy Manual for details on the Tally.

Simulation with SimPy

To use the SimPy simulation system in your Python program you must import its `Simulation` module using:

```
from SimPy.Simulation import *
```

We recommend that new users instead import `SimPy.SimulationTrace`, which works the same but also automatically produces a timed listing of events as the model executes. (An example of such a trace is shown in *The Resource Example with Tracing*):

```
from SimPy.SimulationTrace import *
```

Discrete-event simulation programs automatically maintain the current simulation time in a software clock. This cannot be directly changed by the user. In SimPy the current clock value is returned by the `now()` function and you can access this or print it out. At the start of the simulation it is set to 0.0. While the simulation program runs, simulation time steps forward from one *event* to the next. An event occurs whenever the state of the simulated system changes such as when a car arrives or departs from the gas station.

The `initialize` statement initialises global simulation variables and sets the software clock to 0.0. It must appear in your program before any SimPy process objects are activated.

```
initialize()
```

This is followed by SimPy statements creating and activating entities (that is, SimPy process objects). Activation of entities adds events to the simulation event schedule. Execution of the simulation itself starts with the following statement:

```
simulate(until=endtime)
```

The simulation then starts, and SimPy seeks and executes the first event in the schedule. Having executed that event, the simulation seeks and executes the next event, and so on.

Typically a simulation terminates when there are no more events to execute or when the *endtime* is reached but it can be stopped at any time by the command:

```
stopSimulation( )
```

After the simulation stops, further statements can be executed. `now()` will retain the time of stopping and data held in Monitors will be available for display or further analysis.

The following fragment shows only the *main* block in a simulation program to illustrate the general structure. A complete *Example Program* is shown later. Here `Car` is a Process class with a `go` as its PEM (described later) and `c`

is made an entity of that class, that is, a particular car. Activating `c` has the effect of scheduling at least one event by starting `c`'s PEM. The `simulate(until=1000.0)` statement starts the simulation itself. This immediately jumps to the first scheduled event. It will continue executing events until it runs out of events to execute or the simulation time reaches `1000.0`. When the simulation stops the `Report` function is called to display the results:

```
class Car(Process):
   def go(self):
      # PEM for a Car
      ...

def Report():
   # print results when finished
   ...

initialize()
c = Car(name="Car23")
activate(c, c.go(), at=0.0)
simulate(until=1000.0)

Report()
```

In addition to *SimPy.Simulation* there are three alternative simulation libraries with the same simulation capabilities but with special facilities. Beside `SimPy.SimulationTrace`, already mentioned, there are `SimPy.SimulationRT` for real time synchronisation and `SimPy.SimulationStep` for event-stepping through a simulation. See the SimPy Manual for more information.

Processes

SimPy's active objects (entities) are process objects – instances of a class written by the user that inherits from SimPy's Process class.

For example, if we are simulating a gas station we might model each car as an object of the class `Car`. A car arrives at the gas station (modelled as a Resource with `pumps`). It requests a pump and may need to wait for it. Then it fills its tank and when it has finished it releases the pump. It might also buy an item from the station store. The `Car` class specifies the logic of these actions in its Process Execution Method (PEM). The simulation creates a number of cars as it runs and their evolutions are directed by their `Car` class's PEM.

Defining a process

Each Process class inherits from SimPy's `Process` class. For example the header of the definition of a `Car` Process class would be:

```
class Car(Process):
```

At least one Process Execution Method (PEM) must be defined in each Process class (though an entity can have only one PEM active). A PEM may have arguments in addition to the required `self` argument needed by all Python class methods. Naturally, other methods and, in particular, an `__init__`, may be defined.

- `A Process Execution Method (PEM)` defines the actions that are performed by its process objects. *Each PEM must contain at least one of the special ''yield'' statements, described later.* This makes the PEM a Python generator function so that it has resumable execution – it can be restarted again after the yield statement without losing its current state. A PEM may have any name of your choice. For example it may be called `execute( )` or `run( )`. However, if a PEM is called `ACTIONS`, SimPy recognises this as a PEM. This can simplify the `start` method as explained below.

The `yield` statements are simulation commands which affect an ongoing life cycle of Process objects. These statements control the execution and synchronisation of multiple processes. They can delay a process, put it to sleep, request a shared resource or provide a resource. They can add new events to the simulation event schedule, cancel existing ones, or cause processes to wait for a change in the simulated system's state.

For example, here is a Process Execution Method, `go(self)`, for the simple `Car` class that does no more than delay for a time. As soon as it is activated it prints out the current time, the car object's name and the word `Starting`. After a simulated delay of 100.0 time units (in the `yield hold, ...` statement) it announces that this car has "Arrived":

```
def go(self):
    print("%s %s %s" % (now(), self.name, 'Starting'))
    yield hold, self, 100.0
    print("%s %s %s" % (now(), self.name, 'Arrived'))
```

A process object's PEM starts execution when the object is activated, provided the `simulate(until=`*endtime*`)` statement has been executed.

- **__init__(self, ...)**, where ... indicates other arguments. This method is optional but is useful to initialise the process object, setting values for its attributes. As for any sub-class in Python, the first line of this method must call the `Process` class's `__init__( )` method in the form:

  ```
  Process.__init__(self)
  ```

  You can then use additional commands to initialise attributes of the Process class's objects. You can also override the standard `name` attribute of the object.

  If present, the `__init__( )` method is always called whenever you create a new process object. If you do not wish to provide for any attributes other than a `name`, the `__init__` method may be dispensed with. An example of an `__init__( )` method is shown in the *Example Program*.

Creating a process object

An entity (process object) is created in the usual Python manner by calling the Class. Process classes have a single attribute, `name` which can be specified even if no `__init__` method is defined. `name` defaults to `'a_process'` unless the user specified a different one.

For example to create a new `Car` object with a name `Car23`:

```
c = Car(name="Car23")
```

Starting SimPy Process Objects

An entity (process object) is "passive" when first created, i.e., it has no events scheduled for it. It must be *activated* to start its Process Execution Method. To do this you can use either the `activate` function or the `start` method of the Process.

activate

Activating an entity by using the SimPy `activate` function:

- `activate(`*p, p.pemname([args])*`[,{at=`*t*`|delay=`*period*`}])`

  activates process object *p*, provides its Process Execution Method *p.pemname( )* with the arguments *args* and possibly assigns values to the other optional parameters. You must choose one (or neither) of `at=`*t* and `delay=`*period*. The default is to activate at the current time (`at=now( )`) and with no delay (`delay=0`).

For example: to activate an entity, `cust` at time 10.0 using its PEM called `lifetime`:

```
cust = Customer()
activate(cust, cust.lifetime(), at=10.0)
```

start

An alternative to the `activate()` function is the `start` method of Process objects:

- *p*.`start(`*p.pemname([args])*`[,{at=`*t*`|delay=`*period*`}])`

  Here *p* is a Process object. Its PEM, *pemname*, can have any identifier (such as `run`, `lifecycle`, etc) and any arguments *args*.

  For example, to activate the process object `cust` using the PEM with identifier `lifetime` at time 10.0 we would use:

```
cust.start(cust.lifetime(), at=10.0)
```

The standard PEM name, ACTIONS

The identifier `ACTIONS` is recognised by SimPy as a PEM name and can be used (or implied) in the `start` method.

- *p*.`start([`*p*.`ACTIONS()] [,{at=`*t*`|delay=`*period*`}])`

  `ACTIONS` *cannot* have parameters. The call *p*.`ACTIONS()` is optional but may make your code clearer.

  For example, to activate the Process object `cust` with a PEM called `ACTIONS` at time 10.0, the following are equivalent (and the second version more convenient):

```
cust.start(cust.ACTIONS(), at=10.0)
cust.start(at=10.0)
```

A reminder: Even activated process objects will not actually start operating until the `simulate()` statement is executed.

Elapsing time in a Process

A *PEM* uses the `yield hold` command to temporarily delay a process object's operations. This might represent a service time for the entity. (Waiting is handled automatically by the resource facilities and is not modelled by `yield hold`)

yield hold

`yield hold, self, t`

Causes the entity to delay *t* time units. After the delay, it continues with the next statement in its PEM. During the `hold` the entity's operations are suspended.

Paradoxically, in the model world, the entity is considered to be *busy* during this simulated time. For example, it might be involved in filling up with gas or driving. In this state it can be interrupted by other entities.

More about Processes

An entity (Process object) can be "put to sleep" or passivated using

```
yield passivate, self
```

(and it can be reactivated by another entity using `reactivate`), or permanently removed from the future event queue by the command `self.cancel()`. Active entities can be `interrupt`ed by other entities. Examine the full SimPy Manual for details.

A SimPy Program

This is a complete SimPy script. We define a `Car` class with a PEM called `go( )`. We also (for interest) define an `__init__( )` method to provide individual cars with an identification name and engine size, `cc`. The `cc` attribute is not used in this very simple example.

Two cars, `p1` and `p2` are created. `p1` and `p2` are activated to start at simulation times 0.6 and 0.0, respectively. Note that these will *not* start in the order they appear in the program listing. `p2` actually starts first in the simulation. Nothing happens until the `simulate(until=200)` statement at which point the event scheduler starts operating by finding the first event to execute. When both cars have finished (at time `6.0+100.0=106.0`) there will be no more events so the simulation will stop:

```
from SimPy.Simulation import Process, activate, hold, initialize, now, simulate


class Car(Process):
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield hold, self, 100.0
        print('%s %s %s' % (now(), self.name, 'Arrived'))


initialize()
c1 = Car('Car1', 2000)                # a new car
activate(c1, c1.go(), at=6.0)     # activate at time 6.0
c2 = Car('Car2', 1600)                # another new car
activate(c2, c2.go())                # activate at time 0
simulate(until=200)
print('Current time is %s' % now())     # will print 106.0
```

Running this program gives the following output:

```
0 Car2 Starting
6.0 Car1 Starting
100.0 Car2 Arrived
106.0 Car1 Arrived
Current time is 106.0
```

If, instead one chose to import `SimPy.SimulateTrace` at the start of the program one would obtain the following output. (The meaning of the phrase `prior :  False` in the first two lines is described in the full SimPy Manual.

prior is an advanced technique for fine control of PEM priorities but seldom affects simulated operations and so normally can be ignored/)

```
0 activate <Car1> at time: 6.0 prior: False
0 activate <Car2> at time: 0 prior: False
0 Car2 Starting
0 hold <Car2> delay: 100.0
6.0 Car1 Starting
6.0 hold <Car1> delay: 100.0
100.0 Car2 Arrived
100.0 <Car2> terminated
106.0 Car1 Arrived
106.0 <Car1> terminated
Current time is 106.0
```

Resources

The three resource facilities provided by SimPy are *Resources*, Levels and Stores. Each models a congestion point where process objects may have to queue up to access resources. This section describes the Resource type of resource facility. Levels and Stores are not treated in this introduction.

An example of queueing for a Resource might be a manufacturing plant in which a Task (modelled as an entity or *Process object*) needs work done by a Machine (modelled as a *Resource object*). If all of the Machines are currently being used, the Task must wait until one becomes free. A SimPy Resource can have a number of identical units, such as a number of identical machine units. An entity obtains a unit of the Resource by requesting it and, when it is finished, releasing it. A Resource maintains a list (the waitQ) of entities that have requested but not yet received one of the Resource's units, and another list (the activeQ) of entities that are currently using a unit. SimPy creates and updates these queues itself – the user can read their values, but should not change them.

Defining a Resource object

A Resource object, r, is established by the following statement:

```
r = Resource(capacity=1,
             name='a_resource',
             unitName='units',
             monitored=False)
```

where

- capacity (positive integer) specifies the total number of identical units in Resource object r.
- name (string) the name for this Resource object (e.g., 'gasStation').
- unitName (string) the name for a unit of the resource (e.g., 'pump').
- monitored (False or True) If set to True, then information is gathered on the sizes of r's waitQ and activeQ, otherwise not.

For example, in the model of a 2-pump gas-station we might define:

```
gasstation = Resource(capacity=2, name='gasStation', unitName='pump')
```

Each Resource object, r, has the following additional attributes:

- r.n, the number of units that are currently free.

- `r.waitQ`, a queue (list) of processes that have requested but not yet received a unit of `r`, so `len(r.waitQ)` is the number of process objects currently waiting.
- `r.activeQ`, a queue (list) of process objects currently using one of the Resource's units, so `len(r.activeQ)` is the number of units that are currently in use.
- `r.waitMon`, the record (made by a `Monitor` whenever `monitored == True`) of the activity in `r.waitQ`. So, for example, `r.waitMon.timeaverage()` is the average number of processes in `r.waitQ`. See *Data Summaries* for an example.
- `r.actMon`, the record (made by a `Monitor` whenever `monitored==True`) of the activity in `r.activeQ`.

Requesting and releasing a unit of a Resource

A process can request and later release a unit of the Resource object, `r`, by using the following yield commands in a Process Execution Method:

yield request

- `yield request,self,r`

Requests a unit of Resource *r*

If a Resource unit is free when the request is made, the requesting entity takes it and moves on to the next statement in its PEM. It is added to the Resource's `activeQ`.

If no Resource unit is available when the request is made, the requesting entity is appended to the Resource's `waitQ` and suspended. The next time a unit becomes available the first entity in the `r.waitQ` takes it and continues its execution.

For example, a `Car` might request a `pump`:

```
yield request, self, gasstation
```

(It is actually requesting a *unit* of the `gasstation`, i.e. a `pump`.) An entity holds a resource unit until it releases it.

Entities can use a priority system for queueing. They can also preempt (that is, interrupt) others already in the system. They can also *renege* from the `waitQ` (that is, abandon the queue if it takes too long). This is achieved by an extension to the `yield request` command. See the main SimPy Manual.

yield release

```
yield release, self, r
```

Releases the unit of `r`.

The entity is removed from `r.activeQ` and continues with its next statement. If, when the unit of `r` is released, another entity is waiting (in `waitQ`) it will take the unit, leave the `waitQ` and move into the `activeQ` and go one with its PEM.

For example the `Car` might release the `pump` unit of the gasstation:

```
yield release, self, gasstation
```

Resource Example

In this complete script, the `gasstation` Resource object is given two resource units (`capacity=2`). Four cars arrive at the times specified in the program (not in the order they are listed). They all request a `pump` and use it for 100 time units:

```
from SimPy.Simulation import (Process, Resource, activate, initialize, hold,
                              now, release, request, simulate)


class Car(Process):
    def __init__(self, name, cc):
        Process.__init__(self, name=name)
        self.cc = cc

    def go(self):
        print('%s %s %s' % (now(), self.name, 'Starting'))
        yield request, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Got a pump'))
        yield hold, self, 100.0
        yield release, self, gasstation
        print('%s %s %s' % (now(), self.name, 'Leaving'))


gasstation = Resource(capacity=2, name='gasStation', unitName='pump')
initialize()
c1 = Car('Car1', 2000)
c2 = Car('Car2', 1600)
c3 = Car('Car3', 3000)
c4 = Car('Car4', 1600)
activate(c1, c1.go(), at=4.0)  # activate at time 4.0
activate(c2, c2.go())          # activate at time 0.0
activate(c3, c3.go(), at=3.0)  # activate at time 3.0
activate(c4, c4.go(), at=3.0)  # activate at time 2.0
simulate(until=300)
print('Current time is %s' % now())
```

This program results in the following output:

```
0 Car2 Starting
0 Car2 Got a pump
3.0 Car3 Starting
3.0 Car3 Got a pump
3.0 Car4 Starting
4.0 Car1 Starting
100.0 Car2 Leaving
100.0 Car4 Got a pump
103.0 Car3 Leaving
103.0 Car1 Got a pump
200.0 Car4 Leaving
203.0 Car1 Leaving
Current time is 203.0
```

And, if we use `SimPy.SimulationTrace` to get an automatic trace we get the result shown in Appendix *The Resource Example with Tracing*. (It is rather long to be inserted here).

Random Number Generation

Simulations usually need random numbers. By design, SimPy does not provide its own random number generators, so users need to import them from some other source. Perhaps the most convenient is the standard Python random module. It can generate random variates from the following continuous distributions: uniform, beta, exponential, gamma, normal, log-normal, Weibull, and vonMises. It can also generate random variates from some discrete distributions. Consult the module's documentation for details. Excellent brief descriptions of these distributions, and many others, can be found in the Wikipedia.

Python's `random` module can be used in two ways: you can import the methods directly or you can import the `Random` class and make your own random objects. In the second method, each object gives a different random number sequence, thus providing multiple random streams as in some other simulation languages such as Simscript and ModSim.

Here the first method is illustrated. A single pseudo-random sequence is used for all calls. You `import` the methods you need from the `random` module. For example:

```
from random import seed, random, expovariate, normalvariate
```

In simulation it is good practice to set the initial `seed` for the pseudo-random sequence at the start of each run. You then have good control over the random numbers used. Replications and comparisons are easier and, together with variance reduction techniques, can provide more accurate estimates. In the following code snippet we set the initial seed to `333555`. `X` and `Y` are pseudo-random variates from the two distributions. Both distributions have the same mean:

```
1 from random import seed, expovariate, normalvariate
2
3 seed(333555)
4 X = expovariate(0.1)
5 Y = normalvariate(10.0, 1.0)
```

Monitors and Recording Simulation Results

A Monitor enables us to observe a particular variable of interest and to hold a time series of its values. It can return a simple data summary either during or at the completion of a simulation run.

For example we might use one Monitor to record the waiting time for each of a series of customers and another to record the total number of customers in the shop. In a discrete-event system the number of customers changes only at arrival or departure events and it is at those events that the number in the shop must be observed. A Monitor provides simple statistics useful either alone or as the start of a more sophisticated statistical analysis.

When Resources are defined, a Monitor can be set up to automatically observe the lengths of each of their queues.

The simpler version, the Tally, is not covered in this document. See the SimPy Manual for more details.

Defining Monitors

The `Monitor` class preserves a complete time-series of the observed data values, `y`, and their associated times, `t`. The data are held in a list of two-item sub-lists, `[t,y]`. Monitors calculate data summaries using this time-series when your program requests them. In long simulations their memory demands may be a disadvantage

To define a new Monitor object:

- `m = Monitor(name='a_Monitor')`

  where `name` is a descriptive name for the Monitor object. The descriptive name is used when the data is graphed or tabulated.

For example, to record the waiting times of cars in the gas station we might use a Monitor:

```
waittimes = Monitor(name='Waiting times')
```

Observing data

Monitors use their `observe` method to record data. Here and in the next section, `m` is a Monitor object:

- `m.observe(y [,t])`

  records the current value of the variable, `y` and time *t* (or the current time, `now()`, if *t* is missing). A Monitor retains the two values as a sub-list `[t,y]`.

For example, using the Monitor in the previous example, we might record the waiting times of the cars as shown in the following fragment of the PEM of a `Car`:

```
1 startwaiting = now()              # start wait
2 yield request, self, gasstation
3 waitingtime = now() - startwaiting # time spent waiting
4
5 waittimes.observe(waitingtime)
```

The first three lines measure the waiting time (from the time of the request to the time the `pump` is obtained). The last records the waiting time in the `waittimes` Monitor.

The data recording can be `reset` to start at any time in the simulation:

- `m.reset([t])`

  resets the observations. The recorded data is re-initialised, and the observation starting time is set to *t*, or to the current simulation time, `now( )`, if *t* is missing.

Data summaries

The following simple data summaries are available from Monitors at any time during or after the simulation run:

- `m[i]` holds the `i`-th observation as a two-item list, *[ti, yi]*
- `m.yseries( )` is a list of the recorded data values, *yi*
- `m.tseries( )` is a list of the recorded times, *ti*
- `m.count( )`, the current number of observations. (This is the same as `len(r)`).
- `m.total( )`, the sum of the `y` values
- `m.mean( )`, the simple numerical average of the observed *y* values, *ignoring the times at which they were made*. This is `m.total( )/m.count( )`.
- `m.var( )` the *sample* variance of the observations, ignoring the times at which they were made. If an unbiased estimate of the *population* variance is desired, the sample variance should be multiplied by *n/(n-1)*, where *n = m.count( )*. In either case the standard deviation is, of course, the square-root of the variance
- `m.timeAverage(`*[t]*`)` the time-weighted average of `y`, calculated from time 0 (or the last time `m.reset(`*[t]*`)` was called) to time *t* (or to the current simulation time, `now()`, if *t* is missing).

  This is intended to measure the average of a quantity that always exists, such as the length of a queue or the amount in a Level[1]. In discrete-event simulation such quantity changes are always instantaneous jumps

[1] `timeAverage` is not intended to measure instantaneous values such as a service time or a waiting time. `m.mean()` is used for that.

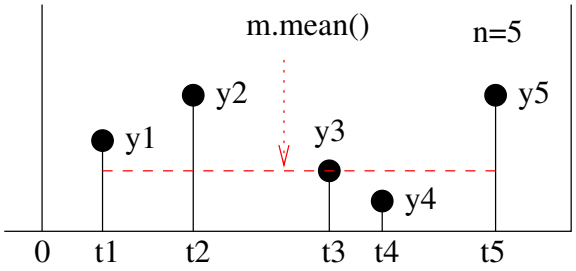


Fig. 2.4: `m.mean` is the simple average of the *y* values observed.

occurring at events. The recorded times and new levels are sufficient information to calculate the average over time. The graph shown in the figure below illustrates the calculation. The total area under the line is calculated and divided by the total time of observation. For accurate time-average results *y* must be piecewise constant like this and observed *just after* each change in its value. That is, the *y* value observed must be the *new* value after the state change.

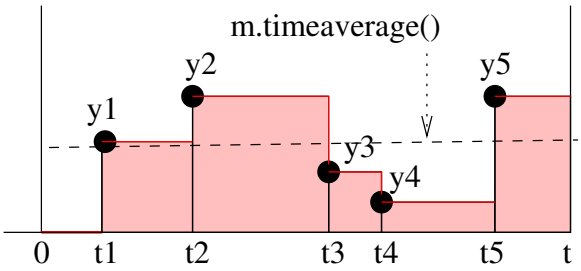


Fig. 2.5: `m.timeAverage([t])` is the time-weighted average of the observed `y` values. Each `y` value is weighted by the time for which it exists. The average is the area under the above curve divided by the total time, *t*.

- `m.timeVariance([t])` the time-weighted variance of the `y` values calculated from time 0 (or the last time `m.reset([t])` was called) to time *t* (or to the current simulation time, `now()`, if *t* is missing).
- `m.__str__( )` is a string that briefly describes the current state of the monitor. This can be used in a print statement.

Monitoring Resource Queues

If a Resource, `m`, (and similarly for a Level or a Store) is defined with `monitored=True`, SimPy automatically records the lengths of its associated queues (i.e. `waitQ` and `activeQ` for Resources, and the analogous queues for Levels and Stores). These records are kept in Monitors `m.waitMon` for the `waitQ` and `m.actMon` for the `activeQ` (and analogously for the other resource types). This solves a problem, particularly for the `waitQ` which cannot easily be recorded externally to the resource.

Complete time series for queue lengths are maintained and can be used for advanced post-simulation statistical analyses and to display summary statistics.

More on Monitors

When a Monitor is defined it is automatically entered into a global list `allMonitors`. Each Monitor also has a descriptive label for its variable values, `y`, and their corresponding times, `t`, that can be used when the data are plotted. The function `startCollection()` can be called to initialise all the Monitors in `allMonitors` at a certain simulation time. This is helpful when a simulation needs a 'warmup' period to achieve steady state before

measurements are started. A Monitor can also generate a `Histogram` of the data (The SimPy `Tally` is an alternative to `Monitor` that does essentially the same job but uses less storage space at some cost of speed and flexibility. See the SimPy Manual for more information on these options.)

Appendices

The Resource Example with Tracing

This is the trace produced in the Resource example when `SimPy.SimulationTrace` is imported at the head of the program. The relevance of the phrases `` `prior: False` `` and `` `priority: default` `` refer to advanced but seldom-needed methods for fine-grained control of event timing, as explained in the SimPy Manual. The trace contains the results of all the `print` output specifically called for by the user's program (for example, line 5) but adds a line for every event executed. (The `request` at time `3.0`, for example, lists the contents of the `waitQ` and the `activeQ` for the `gasstation`.)

```
1  0 activate <Car1> at time: 4.0 prior: False
2  0 activate <Car2> at time: 0 prior: False
3  0 activate <Car3> at time: 3.0 prior: False
4  0 activate <Car4> at time: 3.0 prior: False
5  0 Car2 Starting
6  0 request <Car2> <gasStation> priority: default
7  . . .waitQ: []
8  . . .activeQ: ['Car2']
9  0 Car2 Got a pump
10 0 hold <Car2> delay: 100.0
11 3.0 Car3 Starting
12 3.0 request <Car3> <gasStation> priority: default
13 . . .waitQ: []
14 . . .activeQ: ['Car2', 'Car3']
15 3.0 Car3 Got a pump
16 3.0 hold <Car3> delay: 100.0
17 3.0 Car4 Starting
18 3.0 request <Car4> <gasStation> priority: default
19 . . .waitQ: ['Car4']
20 . . .activeQ: ['Car2', 'Car3']
21 4.0 Car1 Starting
22 4.0 request <Car1> <gasStation> priority: default
23 . . .waitQ: ['Car4', 'Car1']
24 . . .activeQ: ['Car2', 'Car3']
25 100.0 reactivate <Car4> time: 100.0 prior: 1
26 100.0 release <Car2> <gasStation>
27 . . .waitQ: ['Car1']
28 . . .activeQ: ['Car3', 'Car4']
29 100.0 Car2 Leaving
30 100.0 <Car2> terminated
31 100.0 Car4 Got a pump
32 100.0 hold <Car4> delay: 100.0
33 103.0 reactivate <Car1> time: 103.0 prior: 1
34 103.0 release <Car3> <gasStation>
35 . . .waitQ: []
36 . . .activeQ: ['Car4', 'Car1']
37 103.0 Car3 Leaving
38 103.0 <Car3> terminated
39 103.0 Car1 Got a pump
40 103.0 hold <Car1> delay: 100.0
41 200.0 release <Car4> <gasStation>
```

```
. . .waitQ: []
. . .activeQ: ['Car1']
200.0 Car4 Leaving
200.0 <Car4> terminated
203.0 release <Car1> <gasStation>
. . .waitQ: []
. . .activeQ: []
203.0 Car1 Leaving
203.0 <Car1> terminated
Current time is 203.0
```

2.9.2 Cheatsheets

If you want to have a list of all SimPy Classic commands, their syntax and parameters for your desktop, print out a copy of the **SimPy Classic Cheatsheet**, or better yet bookmark them in your favorite browser.

The Cheatsheet comes as a PDF or Excel:

- PDF A4 page
- Excel A4 page

2.9.3 Acknowledgments

SimPy 2.3.3 has been primarily developed by Stefan Scherfke and Ontje Lünsdorf, starting from SimPy 1.9. Their work has resulted in a most elegant combination of an object oriented API with the existing API, maintaining full backward compatibility. It has been quite easy to integrate their product into the existing SimPy code and documentation environment.

Thanks, guys, for this great job! **SimPy 2.0 is dedicated to you!**

The many contributions of the SimPy user and developer communities are of course also gratefully acknowledged.

2.9.4 Indices and tables

- genindex
- search

CHAPTER 3

SimPy Classic Tutorials

There are two styles of modelling using SimPy. The first, which we call the Classical style, uses user-defined Process objects each representing an active entity of the simulation. The simulation is started by creating one or more entities, activating their Process Execution Methods and then, in a main block of the program, calling the `simulate()` function to start the simulation. This simpler style seems to be more acceptable to many users.

In the second style, referred to as the OO Style, a user-defined object of a Simulation class is created and executed. This object executes the whole simulation. This is particularly suitable for running replications of the simulation and the execution of a simulation object is independent of others.

3.1 The Bank

3.1.1 Introduction

In this tutorial, SimPy is used to develop a simple simulation of a bank with a number of tellers. This Python package provides *Processes* to model active components such as messages, customers, trucks, and planes. It has three classes to model facilities where congestion might occur: *Resources* for ordinary queues, *Levels* for the supply of quantities of material, and *Stores* for collections of individual items. Only examples of *Resources* are described here. It also provides *Monitors* and *Tallys* to record data like queue lengths and delay times and to calculate simple averages. It uses the standard Python random package to generate random numbers.

Starting with SimPy 2.0 an object-oriented programmer's interface was added to the package. It is compatible with the current procedural approach which is used in most of the models described here.

SimPy can be obtained from: https://github.com/SimPyClassic/SimPyClassic. The examples run with SimPy version 1.5 and later. This tutorial is best read with the SimPy Manual or Cheatsheet at your side for reference.

Before attempting to use SimPy you should be familiar with the Python language. In particular you should be able to use *classes*. Python is free and available for most machine types. You can find out more about it at the Python web site. SimPy is compatible with Python version 2.7 and Python 3.x.

3.1.2 A customer arriving at a fixed time

In this tutorial we model a simple bank with customers arriving at random. We develop the model step-by-step, starting out simply, and producing a running program at each stage. The programs we develop are available without line numbers and ready to go, in the `bankprograms` directory. Please copy them, run them and improve them - and in the tradition of open-source software suggest your modifications to the SimPy users list. Object-Oriented versions of all the models are included in the bankprograms_OO sub directory.

A simulation should always be developed to answer a specific question; in these models we investigate how changing the number of bank servers or tellers might affect the waiting time for customers.

We first model a single customer who arrives at the bank for a visit, looks around at the decor for a time and then leaves. There is no queueing. First we will assume his arrival time and the time he spends in the bank are fixed.

We define a `Customer` class derived from the SimPy `Process` class. We create a `Customer` object, `c` who arrives at the bank at simulation time `5.0` and leaves after a fixed time of `10.0` minutes.

Examine the following listing which is a complete runnable Python script. We use comments to divide the script up into sections. This makes for clarity later when the programs get more complicated. At #1 is a normal Python documentation string; #2 imports the SimPy simulation code.

The `Customer` class definition at #3 defines our customer class and has the required generator method (called `visit` #4) having a `yield` statement #6. Such a method is called a Process Execution Method (PEM) in SimPy.

The customer's `visit` PEM at #4, models his activities. When he arrives (it will turn out to be a 'he' in this model), he will print out the simulation time, `now()`, and his name at #5. The function `now()` can be used at any time in the simulation to find the current simulation time though it cannot be changed by the programmer. The customer's name will be set when the customer is created later in the script at #10.

He then stays in the bank for a fixed simulation time `timeInBank` at #6. This is achieved by the `yield hold, self,timeInBank` statement. This is the first of the special simulation commands that `SimPy` offers.

After a simulation time of `timeInBank`, the program's execution returns to the line after the `yield` statement at #6. The customer then prints out the current simulation time and his name at #7. This completes the declaration of the `Customer` class.

The call `initialize()` at #9 sets up the simulation system ready to receive `activate` calls. At #10, we create a customer, `c`, with name `Klaus`. All SimPy Processes have a `name` attribute. We `activate Klaus` at #11 specifying the object (`c`) to be activated, the call of the action routine (`c.visit(timeInBank = 10.0)`) and that it is to be activated at time 5 (`at = 5.0`). This will activate `Klaus` exactly `5` minutes after the current time, in this case after the start of the simulation at `0.0`. The call of an action routine such as `c.visit` can specify the values of arguments, here the `timeInBank`.

Finally the call of `simulate(until=maxTime)` at #12 will start the simulation. This will run until the simulation time is `maxTime` unless stopped beforehand either by the `stopSimulation()` command or by running out of events to execute (as will happen here). `maxTime` was set to `100.0` at #8.

```
""" bank01: The single non-random Customer """  # 1
from SimPy.Simulation import *  # 2

# Model components ------------------------------


class Customer(Process):  # 3
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):  # 4
        print("%2.1f %s  Here I am" % (now(), self.name))  # 5
        yield hold, self, timeInBank  # 6
        print("%2.1f %s  I must leave" % (now(), self.name))  # 7
```

```
# Experiment data ------------------------------


maxTime = 100.0      # minutes #8
timeInBank = 10.0    # minutes

# Model/Experiment ------------------------------

initialize()  # 9
c = Customer(name="Klaus")  # 10
activate(c, c.visit(timeInBank), at=5.0)  # 11
simulate(until=maxTime)  # 12
```

The short trace printed out by the `print` statements shows the result. The program finishes at simulation time `15.0` because there are no further events to be executed. At the end of the `visit` routine, the customer has no more actions and no other objects or customers are active.

```
5.0 Klaus  Here I am
15.0 Klaus  I must leave
```

3.1.3 The Bank in object-oriented style

Now look at the same model developed in object-oriented style. As before `Klaus` arrives at the bank for a visit, looks around at the decor for a time and then leaves. There is no queueing. The arrival time and the time he spends in the bank are fixed.

The key point is that we create a `Simulation` object and run that. In the program at #1 we import a new class, `Simulation` together with the familiar `Process` class, and the `hold` verb. Here we are using the recommended explicit form of import rather than the deprecated `import *`.

Just as before, we define a `Customer` class, derived from the SimPy `Process` class, which has the required generator method (PEM), here called `visit`.

The customer's `visit` PEM models his activities. When he arrives at the bank Klaus will print out both the current simulation time, `self.sim.now()`, and his name, `self.name`. The prefix `self.sim` is a reference to the simulation object where this customer exists, thus `self.sim.now()` refers to the clock for that simulation object. Every `Process` instance is linked to the simulation in which it is created by assigning to its `sim` parameter when it is created (at #4).

Now comes the major difference from the Classical SimPy program structure. We define a class `BankModel` from the `Simulation` class at #3. Now any instance of `BankModel` is an independent simulation with its own event list and its own time axis. A `BankModel` instance can activate processes and start the execution of a simulation on its time axis.

In the `BankModel` class, we define a `run` method which, when executes the `BankModel` instance, i.e. performs a simulation experiment. When it starts it initializes the simulation with it event list and sets the time to 0.

#4 creates a `Customer` object and the parameter assignment `sim = self` ties the customer instance to this and only this simulation. The customer does not exist outside this simulation. The call of `simulate(until=maxTime)` at #5 starts the simulation. It will run until the simulation time is `maxTime` unless stopped beforehand either by the `stopSimulation()` command or by running out of events to execute (as will happen here). `maxTime` is set to `100.0` at #6.

Note: If model classes like the `BankModel` are to be given any other attributes, they must have an `__init__`

method in which these attributes are assigned. Such an `__init__` method must first call `Simulation.__init__(self)` to also initialize the `Simulation` class from which the model inherits.

A new, independent simulation object, `mymodel`, is created at #7. Its `run` method is executed at #8.

```
""" bank01_OO: The single non-random Customer """
from SimPy.Simulation import Simulation, Process, hold  # 1
# Model components ------------------------------


class Customer(Process):  # 2
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):
        print("%2.1f %s  Here I am" %
              (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%2.1f %s  I must leave" %
              (self.sim.now(), self.name))

# Model -----------------------------------------


class BankModel(Simulation):  # 3
    def run(self):
        """ An independent simulation object """
        c = Customer(name="Klaus", sim=self)  # 4
        self.activate(c, c.visit(timeInBank), at=tArrival)
        self.simulate(until=maxTime)  # 5


# Experiment data -------------------------------
maxTime = 100.0     # minutes                          #6
timeInBank = 10.0   # minutes
tArrival = 5.0      # minutes

# Experiment ------------------------------------
mymodel = BankModel()  # 7
mymodel.run()  # 8
```

The short trace printed out by the `print` statements shows the result. The program finishes at simulation time `15.0` because there are no further events to be executed. At the end of the `visit` routine, the customer has no more actions and no other objects or customers are active.

```
5.0 Klaus  Here I am
15.0 Klaus  I must leave
```

All of The Bank programs have been written in both the procedural and object orientated styles.

3.1.4 A customer arriving at random

Now we extend the model to allow our customer to arrive at a random simulated time though we will keep the time in the bank at 10.0, as before.

The change occurs at #1, #2, #3, and #4 in the program. At #1 we import from the standard Python `random` module to give us `expovariate` to generate the random time of arrival. We also import the `seed` function to initialize

the random number stream to allow control of the random numbers. At #2 we provide an initial seed of `99999`. An exponential random variate, `t`, is generated at #3. Note that the Python Random module's `expovariate` function uses the average rate (that is, `1.0/mean`) as the argument. The generated random variate, `t`, is used at #4 as the `at` argument to the `activate` call.

```
""" bank05: The single Random Customer """
from SimPy.Simulation import *
from random import expovariate, seed  # 1

# Model components ------------------------


class Customer(Process):
    """ Customer arrives at a random time,
        looks around and then leaves """

    def visit(self, timeInBank):
        print("%f %s Here I am" % (now(), self.name))
        yield hold, self, timeInBank
        print("%f %s I must leave" % (now(), self.name))

# Experiment data -------------------------


maxTime = 100.0    # minutes
timeInBank = 10.0
# Model/Experiment ------------------------------

seed(99999)  # 2
initialize()
c = Customer(name="Klaus")
t = expovariate(1.0 / 5.0)  # 3
activate(c, c.visit(timeInBank), at=t)  # 4
simulate(until=maxTime)
```

The result is shown below. The customer now arrives at about 0.64195, (or 10.58092 if you are not using Python 3). Changing the seed value would change that time.

```
0.641954 Klaus Here I am
10.641954 Klaus I must leave
```

The display looks pretty untidy. In the next example I will try and make it tidier.

If you are not using Python 3, your output may differ. The output for Python 2, for all examples, is given in Appendix A.

3.1.5 More customers

Our simulation does little so far. To consider a simulation with several customers we return to the simple deterministic model and add more `Customers`.

The program is almost as easy as the first example (*A Customer arriving at a fixed time*). The main change is between #4 to #5 where we create, name, and activate three customers. We also increase the maximum simulation time to `400` (at #3 and referred to at #6). Observe that we need only one definition of the `Customer` class and create several objects of that class. These will act quite independently in this model.

Each customer stays for a different `timeinbank` so, instead of setting a common value for this we set it for each customer. The customers are started at different times (using `at=`). `Tony's` activation time occurs before `Klaus's`,

so Tony will arrive first even though his activation statement appears later in the script.

As promised, the print statements have been changed to use Python string formatting (at #1 and #2). The statements look complicated but the output is much nicer.

```
""" bank02: More Customers """
from SimPy.Simulation import *

# Model components ------------------------


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):
        print("%7.4f %s: Here I am" % (now(), self.name))   # 1
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (now(), self.name))   # 2

# Experiment data -------------------------


maxTime = 400.0  # minutes                                    #3

# Model/Experiment ------------------------------

initialize()

c1 = Customer(name="Klaus")   # 4
activate(c1, c1.visit(timeInBank=10.0), at=5.0)
c2 = Customer(name="Tony")
activate(c2, c2.visit(timeInBank=7.0), at=2.0)
c3 = Customer(name="Evelyn")
activate(c3, c3.visit(timeInBank=20.0), at=12.0)   # 5

simulate(until=maxTime)   # 6
```

The trace produced by the program is shown below. Again the simulation finishes before the 400.0 specified in the simulate call as it has run out of events.

```
 2.0000 Tony: Here I am
 5.0000 Klaus: Here I am
 9.0000 Tony: I must leave
12.0000 Evelyn: Here I am
15.0000 Klaus: I must leave
32.0000 Evelyn: I must leave
```

3.1.6 Many customers

Another change will allow us to have more customers. As it is tedious to give a specially chosen name to each one, we will call them Customer00, Customer01,... and use a separate Source class to create and activate them. To make things clearer we do not use the random numbers in this model.

The following listing shows the new program. At #1 to #4 a Source class is defined. Its PEM, here called generate, is defined between #2 to #4. This PEM has a couple of arguments: the number of customers to be generated and the Time Between Arrivals, TBA. It consists of a loop that creates a sequence of numbered Customers from 0 to (number-1), inclusive. We create a customer and give it a name at #3. It is then activated at the current

simulation time (the final argument of the `activate` statement is missing so that the default value of `now()` is used as the time). We also specify how long the customer is to stay in the bank. To keep it simple, all customers stay exactly `12` minutes. After each new customer is activated, the `Source` holds for a fixed time (`yield hold,self,TBA`) before creating the next one (at #4).

A `Source`, `s`, is created at #5 and activated at #6 where the number of customers to be generated is set to `maxNumber = 5` and the interval between customers to `ARRint = 10.0`. Once started at time `0.0` it creates customers at intervals and each customer then operates independently of the others:

```
""" bank03: Many non-random Customers """
from SimPy.Simulation import *

# Model components ------------------------


class Source(Process):  # 1
    """ Source generates customers regularly """

    def generate(self, number, TBA):  # 2
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))  # 3
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, TBA  # 4


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):
        print("%7.4f %s: Here I am" % (now(), self.name))
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (now(), self.name))

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0  # minutes
ARRint = 10.0    # time between arrivals, minutes

# Model/Experiment ------------------------------

initialize()
s = Source()  # 5
activate(s, s.generate(number=maxNumber,  # 6
                       TBA=ARRint), at=0.0)
simulate(until=maxTime)
```

The output is:

```
 0.0000 Customer00: Here I am
10.0000 Customer01: Here I am
12.0000 Customer00: I must leave
20.0000 Customer02: Here I am
22.0000 Customer01: I must leave
30.0000 Customer03: Here I am
32.0000 Customer02: I must leave
40.0000 Customer04: Here I am
42.0000 Customer03: I must leave
```

```
52.0000 Customer04: I must leave
```

3.1.7 Many random customers

We now extend this model to allow arrivals at random. In simulation this is usually interpreted as meaning that the times between customer arrivals are distributed as exponential random variates. There is little change in our program, we use a `Source` object, as before.

The exponential random variate is generated at #1 with `meanTBA` as the mean Time Between Arrivals and used at #2. Note that this parameter is not exactly intuitive. As already mentioned, the Python `expovariate` method uses the *rate* of arrivals as the parameter not the average interval between them. The exponential delay between two arrivals gives pseudo-random arrivals. In this model the first customer arrives at time `0.0`.

The `seed` method is called to initialize the random number stream in the `model` routine (at #3). It is possible to leave this call out but if we wish to do serious comparisons of systems, we must have control over the random variates and therefore control over the seeds. Then we can run identical models with different seeds or different models with identical seeds. We provide the seeds as control parameters of the run. Here a seed is assigned at #3 but it is clear it could have been read in or manually entered on an input form.

```
""" bank06: Many Random Customers """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers at random """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            t = expovariate(1.0 / meanTBA)   # 1
            yield hold, self, t   # 2


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank=0):
        print("%7.4f %s: Here I am" % (now(), self.name))
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (now(), self.name))

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0  # minutes
ARRint = 10.0    # mean arrival interval, minutes

# Model/Experiment ------------------------------

seed(99999)  # 3
initialize()
```

```
s = Source(name='Source')
activate(s, s.generate(number=maxNumber,
                       meanTBA=ARRint), at=0.0)
simulate(until=maxTime)
```

with the following output:

```
 0.0000 Customer00: Here I am
 1.2839 Customer01: Here I am
 4.9842 Customer02: Here I am
12.0000 Customer00: I must leave
13.2839 Customer01: I must leave
16.9842 Customer02: I must leave
35.5432 Customer03: Here I am
47.5432 Customer03: I must leave
48.9918 Customer04: Here I am
60.9918 Customer04: I must leave
```

3.2 Service Counters

We introduce a service counter at the Bank using a SimPy Resource.

3.2.1 A service counter

So far, the model has been more like an art gallery, the customers entering, looking around, and leaving. Now they are going to require service from the bank clerk. We extend the model to include a service counter which will be modelled as an object of SimPy's `Resource` class with a single resource unit. The actions of a `Resource` are simple: a customer `requests` a unit of the resource (a clerk). If one is free he gets service (and removes the unit). If there is no free clerk the customer joins the queue (managed by the resource object) until it is their turn to be served. As each customer completes service and `releases` the unit, the clerk can start serving the next in line.

The service counter is created as a `Resource` (`k`) in at #8. This is provided as an argument to the `Source` (at #9) which, in turn, provides it to each customer it creates and activates (at #1).

The actions involving the service counter, `k`, in the customer's PEM are:

- the `yield request` statement at #3. If the server is free then the customer can start service immediately and the code moves on to #4. If the server is busy, the customer is automatically queued by the Resource. When it eventually comes available the PEM moves on to #4.
- the `yield hold` statement at #5 where the operation of the service counter is modelled. Here the service time is a fixed `timeInBank`. During this period the customer is being served.
- the `yield release` statement at #6. The current customer completes service and the service counter becomes available for any remaining customers in the queue.

Observe that the service counter is used with the pattern (`yield request..; yield hold..; yield release..`).

To show the effect of the service counter on the activities of the customers, I have added #2 to record when the customer arrived and #4 to record the time between arrival in the bank and starting service. #4 is *after* the `yield request` command and will be reached only when the request is satisfied. It is *before* the `yield hold` that corresponds to the start of service. The variable `wait` will record how long the customer waited and will be 0 if he received service at once. This technique of saving the arrival time in a variable is common. So the `print` statement also prints out how long the customer waited in the bank before starting service.

```
""" bank07: One Counter,random arrivals """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0,
                                res=resource))  # 1
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
```

```
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank, res):
        arrive = now()  # 2  arrival time
        print("%8.3f %s: Here I am" % (now(), self.name))

        yield request, self, res  # 3
        wait = now() - arrive  # 4  waiting time
        print("%8.3f %s: Waited %6.3f" % (now(), self.name, wait))
        yield hold, self, timeInBank  # 5
        yield release, self, res  # 6

        print("%8.3f %s: Finished" % (now(), self.name))

# Experiment data -------------------------


maxNumber = 5  # 7
maxTime = 400.0   # minutes
ARRint = 10.0     # mean, minutes
k = Resource(name="Counter", unitName="Clerk")  # 8

# Model/Experiment ------------------------------
seed(99999)
initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber,
                       meanTBA=ARRint, resource=k), at=0.0)  # 9
simulate(until=maxTime)
```

Examining the trace we see that the first, and last, customers get instant service but the others have to wait. We still only have five customers (#7) so we cannot draw general conclusions.

```
   0.000 Customer00: Here I am
   0.000 Customer00: Waited  0.000
   1.284 Customer01: Here I am
   4.984 Customer02: Here I am
  12.000 Customer00: Finished
  12.000 Customer01: Waited 10.716
  24.000 Customer01: Finished
  24.000 Customer02: Waited 19.016
  35.543 Customer03: Here I am
  36.000 Customer02: Finished
  36.000 Customer03: Waited  0.457
  48.000 Customer03: Finished
  48.992 Customer04: Here I am
  48.992 Customer04: Waited  0.000
  60.992 Customer04: Finished
```

3.2.2 A server with a random service time

This is a simple change to the model in that we retain the single service counter but make the customer service time a random variable. As is traditional in the study of simple queues we first assume an exponential service time and set the mean to `timeInBank`.

The service time random variable, `tib`, is generated at #1 and used at #2. The argument to be used in the call of

`expovariate` is not the mean of the distribution, `timeInBank`, but is the rate `1/timeInBank`.

We have also collected together a number of constants by defining a number of appropriate variables and giving them values. These are in lines between marks #3 and #4.

```
""" bank08: A counter with a random service time """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i,))
            activate(c, c.visit(b=resource))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = now()
        print("%8.4f %s: Here I am     " % (now(), self.name))
        yield request, self, b
        wait = now() - arrive
        print("%8.4f %s: Waited %6.3f" % (now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)  # 1
        yield hold, self, tib  # 2
        yield release, self, b
        print("%8.4f %s: Finished      " % (now(), self.name))

# Experiment data -------------------------


maxNumber = 5  # 3
maxTime = 400.0    # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
theseed = 99999  # 4

# Model/Experiment ------------------------------

seed(theseed)
k = Resource(name="Counter", unitName="Clerk")

initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber, meanTBA=ARRint,
                       resource=k), at=0.0)
simulate(until=maxTime)
```

And the output:

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Waited  0.000
  1.2839 Customer01: Here I am
  4.4403 Customer00: Finished
  4.4403 Customer01: Waited  3.156
 20.5786 Customer01: Finished
 31.8430 Customer02: Here I am
 31.8430 Customer02: Waited  0.000
 34.5594 Customer02: Finished
 36.2308 Customer03: Here I am
 36.2308 Customer03: Waited  0.000
 41.4313 Customer04: Here I am
 67.1315 Customer03: Finished
 67.1315 Customer04: Waited 25.700
 87.9241 Customer04: Finished
```

This model with random arrivals and exponential service times is an example of an M/M/1 queue and could rather easily be solved analytically to calculate the steady-state mean waiting time and other operating characteristics. (But not so easily solved for its transient behavior.)

3.2.3 Several service counters

When we introduce several counters we must decide on a queue discipline. Are customers going to make one queue or are they going to form separate queues in front of each counter? Then there are complications - will they be allowed to switch lines (jockey)? We first consider a single queue with several counters and later consider separate isolated queues. We will not look at jockeying.

Here we model a bank whose customers arrive randomly and are to be served at a group of counters, taking a random time for service, where we assume that waiting customers form a single first-in first-out queue.

The *only* difference between this model and the single-server model is at #1. We have provided two counters by increasing the capacity of the `counter` resource to 2. These *units* of the resource correspond to the two counters. Because both clerks cannot be called `Karen`, we have used a general name of `Clerk`.

```
""" bank09: Several Counters but a Single Queue """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(b=resource))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = now()
```

```
        print("%8.4f %s: Here I am      " % (now(), self.name))
        yield request, self, b
        wait = now() - arrive
        print("%8.4f %s: Waited %6.3f" % (now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b
        print("%8.4f %s: Finished      " % (now(), self.name))

# Experiment data ---------------------------


maxNumber = 5
maxTime = 400.0    # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
theseed = 99999

# Model/Experiment ------------------------------

seed(theseed)
k = Resource(capacity=2, name="Counter", unitName="Clerk")  # 1

initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber, meanTBA=ARRint,
                       resource=k), at=0.0)
simulate(until=maxTime)
```

The waiting times in this model are very different from those for the single service counter. For example, none of the customers had to wait. But, again, we have observed too few customers to draw general conclusions.

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Waited  0.000
  1.2839 Customer01: Here I am
  1.2839 Customer01: Waited  0.000
  4.4403 Customer00: Finished
 17.4222 Customer01: Finished
 31.8430 Customer02: Here I am
 31.8430 Customer02: Waited  0.000
 34.5594 Customer02: Finished
 36.2308 Customer03: Here I am
 36.2308 Customer03: Waited  0.000
 41.4313 Customer04: Here I am
 41.4313 Customer04: Waited  0.000
 62.2239 Customer04: Finished
 67.1315 Customer03: Finished
```

3.2.4 Several counters with individual queues

Each counter now has its own queue. The programming is more complicated because the customer has to decide which one to join. The obvious technique is to make each counter a separate resource and it is useful to make a list of resource objects (#7).

In practice, a customer will join the shortest queue. So we define a Python function, `NoInSystem(R)` (#1 to #2) to return the sum of the number waiting and the number being served for a particular counter, `R`. This function is used

at #3 to list the numbers at each counter. It is then easy to find which counter the arriving customer should join. We have also modified the trace printout, #4 to display the state of the system when the customer arrives. We choose the shortest queue at #5 to #6 (using the variable `choice`).

The rest of the program is the same as before.

```
""" bank10: Several Counters with individual queues"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval, counters):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(counters))
            t = expovariate(1.0 / interval)
            yield hold, self, t


def NoInSystem(R):  # 1
    """ Total number of customers in the resource R"""
    return (len(R.waitQ) + len(R.activeQ))  # 2


class Customer(Process):
    """ Customer arrives, chooses the shortest queue
        is served and leaves
    """

    def visit(self, counters):
        arrive = now()
        Qlength = [NoInSystem(counters[i]) for i in range(Nc)]  # 3
        print("%7.4f %s: Here I am. %s" % (now(), self.name, Qlength))  # 4
        for i in range(Nc):  # 5
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                choice = i    # the index of the shortest line
                break  # 6

        yield request, self, counters[choice]
        wait = now() - arrive
        print("%7.4f %s: Waited %6.3f" % (now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counters[choice]

        print("%7.4f %s: Finished" % (now(), self.name))

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0    # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
```

```
Nc = 2              # number of counters
theseed = 787878

# Model/Experiment ------------------------------

seed(theseed)
kk = [Resource(name="Clerk0"), Resource(name="Clerk1")]  # 7
initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber, interval=ARRint,
                       counters=kk), at=0.0)
simulate(until=maxTime)
```

The results show how the customers choose the counter with the smallest number. Unlucky `Customer03` who joins the wrong queue has to wait until `Customer01` finishes before his service can be started. There are, however, too few arrivals in these runs, limited as they are to five customers, to draw any general conclusions about the relative efficiencies of the two systems.

```
 0.0000 Customer00: Here I am. [0, 0]
 0.0000 Customer00: Waited  0.000
 9.7519 Customer00: Finished
12.0829 Customer01: Here I am. [0, 0]
12.0829 Customer01: Waited  0.000
25.9167 Customer02: Here I am. [1, 0]
25.9167 Customer02: Waited  0.000
38.2349 Customer03: Here I am. [1, 1]
40.4032 Customer04: Here I am. [2, 1]
43.0677 Customer02: Finished
43.0677 Customer04: Waited  2.664
44.0242 Customer01: Finished
44.0242 Customer03: Waited  5.789
60.1271 Customer03: Finished
70.2500 Customer04: Finished
```

3.3 Customer's Priority

In Many situations there is a system of priority service. Those customers with high priority are served first, those with low priority must wait. In some cases, preemptive priority will even allow a high-priority customer to interrupt the service of one with a lower priority.

3.3.1 Priority customers

SimPy implements priority requests with an extra numerical priority argument in the `yield request` command, higher values meaning higher priority. For this to operate, the requested Resource must have been defined with `qType=PriorityQ`.

In the first example, we modify the program with random arrivals, one counter, and a fixed service time with the addition of a high priority customer. Warning: The `seed()` value has been changed to `787878` to make the story more exciting. To make things even more confusing, your results may be different from those here because the `random` module gives different results for Python 2.x and 3.x.,

The main modifications are to the definition of the `counter` where we change the `qType` and to the `yield request` command in the `visit` PEM of the customer. We must provide each customer with a priority. Since the default is `priority=0` this is easy for most of them.

To observe the priority in action, while all other customers have the default priority of 0, at between #5 and #6 we create and activate one special customer, `Guido`, with priority 100 who arrives at time `23.0`.

The `visit` customer method has a new parameter, `P` (at #3), which allows us to set the customer priority.

At #4, `counter` is defined with `qType=PriorityQ` so that we can request it with priority (at #3) using the statement `yield request,self,counter,P`

At #2, we now print out the number of customers waiting when each customer arrives.

```
""" bank20: One counter with a priority customer """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0,
                                res=resource, P=0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0, res=None, P=0):  # 1
        arrive = now()        # arrival time
        Nwaiting = len(res.waitQ)
        print("%8.3f %s: Queue is %d on arrival" %  # 2
              (now(), self.name, Nwaiting))
```

```
        yield request, self, res, P  # 3
        wait = now() - arrive  # waiting time
        print("%8.3f %s: Waited %6.3f" %
              (now(), self.name, wait))
        yield hold, self, timeInBank
        yield release, self, res

        print("%8.3f %s: Completed" %
              (now(), self.name))

# Experiment data ------------------------


maxTime = 400.0  # minutes
k = Resource(name="Counter", unitName="Karen",  # 4
             qType=PriorityQ)

# Model/Experiment -----------------------------
seed(787878)
initialize()
s = Source('Source')
activate(s, s.generate(number=5, interval=10.0,
                       resource=k), at=0.0)
guido = Customer(name="Guido     ")  # 5
activate(guido, guido.visit(timeInBank=12.0, res=k,  # 6
                            P=100), at=23.0)
simulate(until=maxTime)
```

The resulting output is as follows. The number of customers in the queue just as each arrives is displayed in the trace. That count does not include any customer in service.

```
   0.000 Customer00: Queue is 0 on arrival
   0.000 Customer00: Waited  0.000
  12.000 Customer00: Completed
  12.083 Customer01: Queue is 0 on arrival
  12.083 Customer01: Waited  0.000
  20.210 Customer02: Queue is 0 on arrival
  23.000 Guido     : Queue is 1 on arrival
  24.083 Customer01: Completed
  24.083 Guido     : Waited  1.083
  34.043 Customer03: Queue is 1 on arrival
  36.083 Guido     : Completed
  36.083 Customer02: Waited 15.873
  48.083 Customer02: Completed
  48.083 Customer03: Waited 14.040
  60.083 Customer03: Completed
  60.661 Customer04: Queue is 0 on arrival
  60.661 Customer04: Waited  0.000
  72.661 Customer04: Completed
```

Reading carefully one can see that when `Guido` arrives `Customer00` has been served and left at `12.000`, `Customer01` is in service and Customer02 is waiting in the queue. `Guido` has priority over any waiting customers and is served before the at `24.083`. When `Guido` leaves at `36.083`, `Customer02` starts service having waited `15.873` minutes.

3.3.2 A priority customer with preemption

Now we allow `Guido` to have preemptive priority. He will displace any customer in service when he arrives. That customer will resume when `Guido` finishes (unless higher priority customers intervene). It requires only a change to one line of the program, adding the argument, `preemptable=True` to the `Resource` statement at #1.

```
""" bank23: One counter with a priority customer with preemption """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0,
                                res=resource, P=0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0, res=None, P=0):
        arrive = now()       # arrival time
        Nwaiting = len(res.waitQ)
        print("%8.3f %s: Queue is %d on arrival" %
              (now(), self.name, Nwaiting))

        yield request, self, res, P
        wait = now() - arrive  # waiting time
        print("%8.3f %s: Waited %6.3f" %
              (now(), self.name, wait))
        yield hold, self, timeInBank
        yield release, self, res

        print("%8.3f %s: Completed" %
              (now(), self.name))

# Experiment data -------------------------


maxTime = 400.0  # minutes
k = Resource(name="Counter", unitName="Karen",  # 1
             qType=PriorityQ, preemptable=True)

# Model/Experiment ------------------------------
seed(989898)
initialize()
s = Source('Source')
activate(s, s.generate(number=5, interval=10.0,
                       resource=k), at=0.0)
guido = Customer(name="Guido     ")
```

```
activate(guido, guido.visit(timeInBank=12.0, res=k,
                            P=100), at=23.0)
simulate(until=maxTime)
```

Though `Guido` arrives at the same time, `23.000`, he no longer has to wait and immediately goes into service, displacing the incumbent, `Customer01`. That customer had already completed `23.000-12.083 = 10.917` minutes of his service. When `Guido` finishes at `36.083`, `Customer01` resumes service and takes `36.083-35.000 = 1.083` minutes to finish. His total service time is the same as before (`12.000` minutes).

```
  0.000 Customer00: Queue is 0 on arrival
  0.000 Customer00: Waited  0.000
  8.634 Customer01: Queue is 0 on arrival
 12.000 Customer00: Completed
 12.000 Customer01: Waited  3.366
 16.016 Customer02: Queue is 0 on arrival
 19.882 Customer03: Queue is 1 on arrival
 20.246 Customer04: Queue is 2 on arrival
 23.000 Guido     : Queue is 3 on arrival
 23.000 Guido     : Waited  0.000
 35.000 Guido     : Completed
 36.000 Customer01: Completed
 36.000 Customer02: Waited 19.984
 48.000 Customer02: Completed
 48.000 Customer03: Waited 28.118
 60.000 Customer03: Completed
 60.000 Customer04: Waited 39.754
 72.000 Customer04: Completed
```

3.4 Balking and Reneging Customers

Balking occurs when a customer refuses to join a queue if it is too long. Reneging (or, better, abandonment) occurs if an impatient customer gives up while still waiting and before being served.

3.4.1 Balking customers

Another term for a system with balking customers is one where "blocked customers" are "cleared", termed by engineers a BCC system. This is very convenient analytically in queueing theory and formulae developed using this assumption are used extensively for planning communication systems. The easiest case is when no queueing is allowed.

As an example let us investigate a BCC system with a single server but the waiting space is limited. We will estimate the rate of balking when the maximum number in the queue is set to 1. On arrival into the system the customer must first check to see if there is room. We will need the number of customers in the system or waiting. We could keep a count, incrementing when a customer joins the queue or, since we have a Resource, use the length of the Resource's `waitQ`. Choosing the latter we test (at #1). If there is not enough room, we balk, incrementing a Class variable `Customer.numBalking` at #2 to get the total number balking during the run.

```
""" bank24. BCC system with several counters """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(b=resource))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    numBalking = 0

    def visit(self, b):
        arrive = now()
        print("%8.4f %s: Here I am " %
              (now(), self.name))
        if len(b.waitQ) < maxInQueue:  # 1
            yield request, self, b
            wait = now() - arrive
            print("%8.4f %s: Wait %6.3f" %
                  (now(), self.name, wait))
            tib = expovariate(1.0 / timeInBank)
            yield hold, self, tib
            yield release, self, b
            print("%8.4f %s: Finished  " %
                  (now(), self.name))
```

```
        else:
            Customer.numBalking += 1  # 2
            print("%8.4f %s: BALKING    " %
                  (now(), self.name))

# Experiment data -------------------------------


timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean interarrival time, minutes
numServers = 1     # servers
maxInSystem = 2    # customers
maxInQueue = maxInSystem - numServers

maxNumber = 8
maxTime = 4000.0  # minutes
theseed = 212121

# Model/Experiment ------------------------------

seed(theseed)
k = Resource(capacity=numServers,
             name="Counter", unitName="Clerk")

initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber,
                       meanTBA=ARRint,
                       resource=k), at=0.0)
simulate(until=maxTime)

# Results --------------------------------------

nb = float(Customer.numBalking)
print("balking rate is %8.4f per minute" % (nb / now()))
```

The resulting output for a run of this program showing balking occurring is given below:

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Wait  0.000
  4.3077 Customer01: Here I am
  5.6957 Customer02: Here I am
  5.6957 Customer02: BALKING
  6.9774 Customer03: Here I am
  6.9774 Customer03: BALKING
  8.2476 Customer00: Finished
  8.2476 Customer01: Wait  3.940
 21.1312 Customer04: Here I am
 22.4840 Customer01: Finished
 22.4840 Customer04: Wait  1.353
 23.0923 Customer05: Here I am
 23.1537 Customer06: Here I am
 23.1537 Customer06: BALKING
 36.0653 Customer04: Finished
 36.0653 Customer05: Wait 12.973
 38.4851 Customer07: Here I am
 53.1056 Customer05: Finished
 53.1056 Customer07: Wait 14.620
```

```
 60.3558 Customer07: Finished
balking rate is   0.0497 per minute
```

When `Customer02` arrives, `Customer00` is already in service and `Customer01` is waiting. There is no room so `Customer02` balks. In fact another customer, `Customer03` arrives and balks before `Customer00` is finished.

The balking pattern for python 2.x is different.

3.4.2 Reneging or abandoning

Often in practice an impatient customer will leave the queue before being served. SimPy can model this *reneging* behaviour using a *compound yield statement*. In such a statement there are two yield clauses. An example is:

```
yield (request,self,counter),(hold,self,maxWaitTime)
```

The first tuple of this statement is the usual `yield request`, asking for a unit of `counter` Resource. The process will either get the unit immediately or be queued by the Resource. The second tuple is a reneging clause which has the same syntax as a `yield hold`. The requesting process will renege if the wait exceeds `maxWaitTime`.

There is a complication, though. The requesting PEM must discover what actually happened. Did the process get the resource or did it renege? This involves a *mandatory* test of `self.acquired(`*resource*`)`. In our example, this test is at #1.

```
""" bank21: One counter with impatient customers """
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=15.0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0):
        arrive = now()        # arrival time
        print("%8.3f %s: Here I am     " % (now(), self.name))

        yield (request, self, counter), (hold, self, maxWaitTime)
        wait = now() - arrive  # waiting time
        if self.acquired(counter):  # 1
            print("%8.3f %s: Waited %6.3f" % (now(), self.name, wait))
            yield hold, self, timeInBank
            yield release, self, counter
            print("%8.3f %s: Completed" % (now(), self.name))
        else:
            print("%8.3f %s: Waited %6.3f. I am off" %
```

```
                  (now(), self.name, wait))

# Experiment data ------------------------


maxTime = 400.0  # minutes
maxWaitTime = 12.0  # minutes. maximum time to wait

# Model  ---------------------------------


def model():
    global counter
    seed(989898)
    counter = Resource(name="Karen")
    initialize()
    source = Source('Source')
    activate(source,
             source.generate(number=5, interval=10.0), at=0.0)
    simulate(until=maxTime)

# Experiment  ----------------------------------


model()
```

```
   0.000 Customer00: Here I am
   0.000 Customer00: Waited  0.000
   8.634 Customer01: Here I am
  15.000 Customer00: Completed
  15.000 Customer01: Waited  6.366
  16.016 Customer02: Here I am
  19.882 Customer03: Here I am
  20.246 Customer04: Here I am
  28.016 Customer02: Waited 12.000. I am off
  30.000 Customer01: Completed
  30.000 Customer03: Waited 10.118
  32.246 Customer04: Waited 12.000. I am off
  45.000 Customer03: Completed
```

`Customer02` arrives at 16.016 but has only 12 minutes patience. After that time in the queue (at time 28.016) he abandons the queue to leave 03 to take his place. 04 also abandons.

The reneging pattern for python 2.x is different due to a different expovariate implementation.

3.5 Gathering Statistics

SimPy Monitors allow statistics to be gathered and simple summaries calculated.

3.5.1 The bank with a monitor

The traces of output that have been displayed so far are valuable for checking that the simulation is operating correctly but would become too much if we simulate a whole day. We do need to get results from our simulation to answer the original questions. What, then, is the best way to summarize the results?

One way is to analyze the traces elsewhere, piping the trace output, or a modified version of it, into a *real* statistical program such as *R* for statistical analysis, or into a file for later examination by a spreadsheet. We do not have space to examine this thoroughly here. Another way of presenting the results is to provide graphical output.

SimPy offers an easy way to gather a few simple statistics such as averages: the `Monitor` and `Tally` classes. The `Monitor` records the values of chosen variables as time series (but see the comments in *Final Remarks*).

We now demonstrate a `Monitor` that records the average waiting times for our customers. We return to the system with random arrivals, random service times and a single queue and remove the old trace statements. In practice, we would make the printouts controlled by a variable, say, `TRACE` which is set in the experimental data (or read in as a program option - but that is a different story). This would aid in debugging and would not complicate the data analysis. We will run the simulations for many more arrivals.

A Monitor, `wM`, is created at #2. It `observes` and records the waiting time mentioned at #1. We run `maxNumber=50` customers (in the call of `generate` at #3) and have increased `maxTime` to `1000` minutes. Brief statistics are given by the Monitor methods `count()` and `mean()` at #4.

```
""" bank11: The bank with a Monitor"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval, resource):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(b=resource))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = now()
        yield request, self, b
        wait = now() - arrive
        wM.observe(wait)  # 1
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b

# Experiment data -------------------------


maxNumber = 50
maxTime = 1000.0    # minutes
timeInBank = 12.0   # mean, minutes
ARRint = 10.0    # mean, minutes
Nc = 2           # number of counters
theseed = 99999

# Model/Experiment   ----------------------
```

```
seed(theseed)
k = Resource(capacity=Nc, name="Clerk")
wM = Monitor()  # 2
initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber, interval=ARRint,
                       resource=k), at=0.0)  # 3
simulate(until=maxTime)

# Result  ---------------------------------

result = wM.count(), wM.mean()  # 4
print("Average wait for %3d completions was %5.3f minutes." % result)
```

The average waiting time for 50 customers in this 2-counter system is more reliable (i.e., less subject to random simulation effects) than the times we measured before but it is still not sufficiently reliable for real-world decisions. We should also replicate the runs using different random number seeds. The result of this run (using Python 3.2) is:

```
Average wait for  50 completions was 8.941 minutes.
```

Result for Python 2.x is given in Appendix A.

3.5.2 Monitoring a resource

Now consider observing the number of customers waiting or executing in a Resource. Because of the need to `observe` the value after the change but at the same simulation instant, it is impossible to use the length of the Resource's `waitQ` directly with a Monitor defined outside the Resource. Instead Resources can be set up with built-in Monitors.

Here is an example using a Monitored Resource. We intend to observe the average number waiting and active in the `counter` resource. `counter` is defined at #1 and we have set `monitored=True`. This establishes two Monitors: `waitMon`, to record changes in the numbers waiting and `actMon` to record changes in the numbers active in the `counter`. We need make no further change to the operation of the program as monitoring is then automatic. No `observe` calls are necessary.

At the end of the run in the `model` function, we calculate the `timeAverage` of both `waitMon` and `actMon` and return them from the `model` call (at #2). These can then be printed at the end of the program (at #3).

```
"""bank15: Monitoring a Resource"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
```

```
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank):
        arrive = now()
        print("%8.4f %s: Arrived     " % (now(), self.name))

        yield request, self, counter
        print("%8.4f %s: Got counter " % (now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        print("%8.4f %s: Finished    " % (now(), self.name))


# Experiment data -------------------------
maxTime = 400.0    # minutes
counter = Resource(1, name="Clerk", monitored=True)  # 1

# Model  ----------------------------------


def model(SEED=393939):
    seed(SEED)
    initialize()
    source = Source()
    activate(source,
             source.generate(number=5, rate=0.1), at=0.0)
    simulate(until=maxTime)

    return (counter.waitMon.timeAverage(),
            counter.actMon.timeAverage())  # 2
# Experiment  ----------------------------------


print('Avge waiting = %6.4f\nAvge active = %6.4f\n' % model())  # 3
```

With the following output:

```
  0.0000 Customer00: Arrived
  0.0000 Customer00: Got counter
  1.1489 Customer01: Arrived
  5.6657 Customer02: Arrived
 10.0126 Customer00: Finished
 10.0126 Customer01: Got counter
 12.8923 Customer03: Arrived
 18.5883 Customer04: Arrived
 29.3088 Customer01: Finished
 29.3088 Customer02: Got counter
 44.1219 Customer02: Finished
 44.1219 Customer03: Got counter
 73.0066 Customer03: Finished
 73.0066 Customer04: Got counter
 73.8458 Customer04: Finished
Avge waiting = 1.6000
Avge active = 1.0000
```

3.5.3 Multiple runs

To get a number of independent measurements we must replicate the runs using different random number seeds. Each replication must be independent of previous ones so the Monitor and Resources must be redefined for each run. We can no longer allow them to be global objects as we have before.

We will define a function, `model` with a parameter `runSeed` so that the random number seed can be different for different runs (between between #2 and #5). The contents of the function are the same as the `Model/Experiment` in bank11: *The bank with a monitor*, except for one vital change.

This is required since the Monitor, `wM`, is defined inside the `model` function (#3). A customer can no longer refer to it. In the spirit of quality computer programming we will pass `wM` as a function argument. Unfortunately we have to do this in two steps, first to the `Source` (#4) and then from the `Source` to the `Customer` (#1).

`model()` is run for four different random-number seeds to get a set of replications (between #6 and #7).

```
""" bank12: Multiple runs of the bank with a Monitor"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval, resource, mon):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(b=resource, M=mon))   # 1
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b, M):
        arrive = now()
        yield request, self, b
        wait = now() - arrive
        M.observe(wait)
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b

# Experiment data -------------------------


maxNumber = 50
maxTime = 2000.0  # minutes
timeInBank = 12.0   # mean, minutes
ARRint = 10.0     # mean, minutes
Nc = 2            # number of counters
theSeed = 393939

# Model  ----------------------------------
```

```
def model(runSeed=theSeed):  # 2
    seed(runSeed)
    k = Resource(capacity=Nc, name="Clerk")
    wM = Monitor()  # 3

    initialize()
    s = Source('Source')
    activate(s, s.generate(number=maxNumber, interval=ARRint,
                           resource=k, mon=wM), at=0.0)  # 4
    simulate(until=maxTime)
    return (wM.count(), wM.mean())  # 5

# Experiment/Result  ---------------------------------


theseeds = [393939, 31555999, 777999555, 319999771]  # 6
for Sd in theseeds:
    result = model(Sd)
    print("Avge wait for %3d completions was %6.2f min." %
          result)  # 7
```

The results show some variation. Remember, though, that the system is still only operating for 50 customers so the system may not be in steady-state.

```
Avge wait for  50 completions was   3.66 min.
Avge wait for  50 completions was   2.62 min.
Avge wait for  50 completions was   8.97 min.
Avge wait for  50 completions was   5.34 min.
```

3.6 Final Remarks

This introduction is too long and the examples are getting longer. There is much more to say about simulation with *SimPy* but no space. I finish with a list of topics for further study:

3.6.1 Topics not yet mentioned

- **GUI input**. Graphical input of simulation parameters could be an advantage in some cases. *SimPy* allows this and programs using these facilities have been developed (see, for example, program `MM1.py` in the examples in the *SimPy* distribution)
- **Graphical Output**. Similarly, graphical output of results can also be of value, not least in debugging simulation programs and checking for steady-state conditions. SimPlot is useful here.
- **Statistical Output**. The `Monitor` class is useful in presenting results but more powerful methods of analysis are often needed. One solution is to output a trace and read that into a large-scale statistical system such as *R*.
- **Priorities and Reneging in queues**. *SimPy* allows processes to request units of resources under a priority queue discipline (preemptive or not). It also allows processes to renege from a queue.
- **Other forms of Resource Facilities**. *SimPy* has two other resource structures: `Levels` to hold bulk commodities, and `Stores` to contain an inventory of different object types.
- **Advanced synchronization/scheduling commands**. *SimPy* allows process synchronization by events and signals.

3.6.2 Acknowledgements

I thank Klaus Muller, Bob Helmbold, Mukhlis Matti, Karen Turner and other developers and users of SimPy for improving this document by sending their comments. I would be grateful for further suggestions or corrections. Please send them to: *vignaux* at *users.sourceforge.net*.

3.6.3 References

- Python website: https://www.python.org
- SimPy Classic website: https://github.com/SimPyClassic/SimPyClassic

3.7 Appendix A

With Python 3 the definition of expovariate changed. In some cases this was back ported to some distributions of Python 2.7. Because of this the output for the bank programs varies. This section contains the older output produced by the original definition of expovariate.

A Customer arriving at a fixed time

```
5.0 Klaus  Here I am
15.0 Klaus  I must leave
```

A Customer arriving at random

```
10.580923 Klaus Here I am
20.580923 Klaus I must leave
```

More Customers

```
 2.0000 Tony: Here I am
 5.0000 Klaus: Here I am
 9.0000 Tony: I must leave
12.0000 Evelyn: Here I am
15.0000 Klaus: I must leave
32.0000 Evelyn: I must leave
```

Many Customers

```
 0.0000 Customer00: Here I am
10.0000 Customer01: Here I am
12.0000 Customer00: I must leave
20.0000 Customer02: Here I am
22.0000 Customer01: I must leave
30.0000 Customer03: Here I am
32.0000 Customer02: I must leave
40.0000 Customer04: Here I am
42.0000 Customer03: I must leave
52.0000 Customer04: I must leave
```

Many Random Customers

```
 0.0000 Customer00: Here I am
12.0000 Customer00: I must leave
21.1618 Customer01: Here I am
```

```
32.8968 Customer02: Here I am
33.1618 Customer01: I must leave
33.3790 Customer03: Here I am
36.3979 Customer04: Here I am
44.8968 Customer02: I must leave
45.3790 Customer03: I must leave
48.3979 Customer04: I must leave
```

A Service Counter

```
  0.000 Customer00: Here I am
  0.000 Customer00: Waited  0.000
 12.000 Customer00: Finished
 21.162 Customer01: Here I am
 21.162 Customer01: Waited  0.000
 32.897 Customer02: Here I am
 33.162 Customer01: Finished
 33.162 Customer02: Waited  0.265
 33.379 Customer03: Here I am
 36.398 Customer04: Here I am
 45.162 Customer02: Finished
 45.162 Customer03: Waited 11.783
 57.162 Customer03: Finished
 57.162 Customer04: Waited 20.764
 69.162 Customer04: Finished
```

A server with a random service time

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
14.0819 Customer00: Finished
21.1618 Customer01: Here I am
21.1618 Customer01: Waited  0.000
21.6441 Customer02: Here I am
24.7845 Customer01: Finished
24.7845 Customer02: Waited  3.140
31.9954 Customer03: Here I am
41.0215 Customer04: Here I am
43.9441 Customer02: Finished
43.9441 Customer03: Waited 11.949
49.9552 Customer03: Finished
49.9552 Customer04: Waited  8.934
52.2900 Customer04: Finished
```

Several Counters but a Single Queue

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
14.0819 Customer00: Finished
21.1618 Customer01: Here I am
21.1618 Customer01: Waited  0.000
21.6441 Customer02: Here I am
21.6441 Customer02: Waited  0.000
24.7845 Customer01: Finished
31.9954 Customer03: Here I am
31.9954 Customer03: Waited  0.000
32.9459 Customer03: Finished
40.8037 Customer02: Finished
```

```
 41.0215 Customer04: Here I am
 41.0215 Customer04: Waited  0.000
 43.3562 Customer04: Finished
```

Several Counters with individual queues

```
 0.0000 Customer00: Here I am. [0, 0]
 0.0000 Customer00: Waited  0.000
 3.5483 Customer01: Here I am. [1, 0]
 3.5483 Customer01: Waited  0.000
 4.4169 Customer01: Finished
 6.4349 Customer02: Here I am. [1, 0]
 6.4349 Customer02: Waited  0.000
 7.0368 Customer00: Finished
 9.7200 Customer02: Finished
 9.8846 Customer03: Here I am. [0, 0]
 9.8846 Customer03: Waited  0.000
19.8340 Customer03: Finished
26.2357 Customer04: Here I am. [0, 0]
26.2357 Customer04: Waited  0.000
29.8709 Customer04: Finished
```

Priority Customers

```
  0.000 Customer00: Queue is 0 on arrival
  0.000 Customer00: Waited  0.000
  3.548 Customer01: Queue is 0 on arrival
  9.412 Customer02: Queue is 1 on arrival
 12.000 Customer00: Completed
 12.000 Customer01: Waited  8.452
 12.299 Customer03: Queue is 1 on arrival
 13.023 Customer04: Queue is 2 on arrival
 23.000 Guido     : Queue is 3 on arrival
 24.000 Customer01: Completed
 24.000 Guido     : Waited  1.000
 36.000 Guido     : Completed
 36.000 Customer02: Waited 26.588
 48.000 Customer02: Completed
 48.000 Customer03: Waited 35.701
 60.000 Customer03: Completed
 60.000 Customer04: Waited 46.977
 72.000 Customer04: Completed
```

A priority Customer with Preemption

```
  0.000 Customer00: Queue is 0 on arrival
  0.000 Customer00: Waited  0.000
  5.477 Customer01: Queue is 0 on arrival
 11.978 Customer02: Queue is 1 on arrival
 12.000 Customer00: Completed
 12.000 Customer01: Waited  6.523
 23.000 Guido     : Queue is 1 on arrival
 23.000 Guido     : Waited  0.000
 23.350 Customer03: Queue is 2 on arrival
 35.000 Guido     : Completed
 36.000 Customer01: Completed
 36.000 Customer02: Waited 24.022
 48.000 Customer02: Completed
```

```
  48.000 Customer03: Waited 24.650
  56.664 Customer04: Queue is 0 on arrival
  60.000 Customer03: Completed
  60.000 Customer04: Waited  3.336
  72.000 Customer04: Completed
```

Balking Customers

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Wait  0.000
  8.3884 Customer00: Finished
 10.4985 Customer01: Here I am
 10.4985 Customer01: Wait  0.000
 30.9314 Customer02: Here I am
 33.7131 Customer03: Here I am
 33.7131 Customer03: BALKING
 35.9122 Customer01: Finished
 35.9122 Customer02: Wait  4.981
 37.3562 Customer04: Here I am
 41.2491 Customer05: Here I am
 41.2491 Customer05: BALKING
 56.6185 Customer02: Finished
 56.6185 Customer04: Wait 19.262
 59.5365 Customer04: Finished
 92.2071 Customer06: Here I am
 92.2071 Customer06: Wait  0.000
 94.9740 Customer07: Here I am
102.7425 Customer06: Finished
102.7425 Customer07: Wait  7.769
133.5005 Customer07: Finished
balking rate is   0.0150 per minute
```

Reneging or Abandoning

```

```

The Bank with a Monitor

```
Average wait for  50 completions was 3.430 minutes.
```

Monitoring a Resource

```
  0.0000 Customer00: Arrived
  0.0000 Customer00: Got counter
  6.8329 Customer00: Finished
 22.2064 Customer01: Arrived
 22.2064 Customer01: Got counter
 30.1802 Customer01: Finished
 32.3277 Customer02: Arrived
 32.3277 Customer02: Got counter
 34.5627 Customer03: Arrived
 42.3374 Customer02: Finished
 42.3374 Customer03: Got counter
 46.4642 Customer03: Finished
 61.7697 Customer04: Arrived
 61.7697 Customer04: Got counter
 94.1098 Customer04: Finished
Avge waiting = 0.0826
```

```
Avge active = 0.6512
```

Multiple runs

```
Avge wait for  50 completions was   2.75 min.
Avge wait for  50 completions was   6.01 min.
Avge wait for  50 completions was   5.53 min.
Avge wait for  50 completions was   3.76 min.
```

3.8 Introduction

The first Bank tutorial, The Bank, developed and explained a series of simulation models of a simple bank using SimPy. In various models, customers arrived randomly, queued up to be served at one or several counters, modelled using the Resource class, and, in one case, could choose the shortest among several queues. It demonstrated the use of the Monitor class to record delays and showed how a `model()` mainline for the simulation was convenient to execute replications of simulation runs.

In this extension to The Bank, I provide more examples of SimPy facilities for which there was no room and for some that were developed since it was written. These facilities are generally more complicated than those introduced before. They include queueing with priority, possibly with preemption, reneging, plotting, interrupting, waiting until a condition occurs (`waituntil`) and waiting for events to occur.

The programs are available without line numbers and ready to go, in directory `bankprograms`. Some have trace statements for demonstration purposes, others produce graphical output to the screen. Let me encourage you to run them and modify them for yourself

SimPy itself can be obtained from: https://github.com/SimPyClassic/SimPyClassic. It is compatible with Python version 2.7 onwards. The examples in this documentation run with SimPy version 1.5 and later.

This tutorial should be read with the SimPy Manual or Cheatsheet at your side for reference.

3.9 Processes

In some simulations it is valuable for one SimPy Process to interrupt another. This can only be done when the *victim* is "active"; that is when it has an event scheduled for it. It must be executing a `yield hold` statement.

A process waiting for a resource (after a `yield request` statement) is passive and cannot be interrupted by another. Instead the `yield waituntil` and `yield waitevent` facilities have been introduced to allow processes to wait for conditions set by other processes.

3.9.1 Interrupting a Process.

`Klaus` goes into the bank to talk to the manager. For clarity we ignore the counters and other customers. During his conversation his cellphone rings. When he finishes the call he continues the conversation.

In this example, `call` is an object of the `Call` Process class whose only purpose is to make the cellphone ring after a delay, `timeOfCall`, an argument to its `ring` PEM (label #7).

`klaus`, a `Customer`, is interrupted by the call (Label #8). He is in the middle of a `yield hold` (label #1). When he exits from that command it is as if he went into a trance when talking to the bank manager. He suddenly wakes up and must check (label #2) to see whether has finished his conversation (if there was no call) or has been interrupted.

If `self.interrupted()` is `False` he was not interrupted and leaves the bank (label #6) normally. If it is `True`, he was interrupted by the call, remembers how much conversation he has left (Label #3), resets the interrupt and then deals with the call. When he finishes (label #4) he can resume the conversation, with, now we assume, a thoroughly irritated bank manager (label #5).

```
""" bank22: An interruption by a phone call """
from SimPy.Simulation import *

# Model components ------------------------


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank, onphone):
        print("%7.4f %s: Here I am" % (now(), self.name))
        yield hold, self, timeInBank  # 1
        if self.interrupted():  # 2
            timeleft = self.interruptLeft  # 3
            self.interruptReset()
            print("%7.4f %s: Excuse me" % (now(), self.name))
            print("%7.4f %s: Hello! I'll call back" % (now(), self.name))
            yield hold, self, onphone
            print("%7.4f %s: Sorry, where were we?" % (now(), self.name))  # 4
            yield hold, self, timeleft  # 5
        print("%7.4f %s: I must leave" % (now(), self.name))  # 6


class Call(Process):
    """ Cellphone call arrives and interrupts """

    def ring(self, klaus, timeOfCall):  # 7
        yield hold, self, timeOfCall  # 8
        print("%7.4f Ringgg!" % (now()))
        self.interrupt(klaus)
```

```
# Experiment data ------------------------


timeInBank = 20.0
timeOfCall = 9.0
onphone = 3.0
maxTime = 100.0


# Model/Experiment  ----------------------------------

initialize()
klaus = Customer(name="Klaus")
activate(klaus, klaus.visit(timeInBank, onphone))
call = Call(name="klaus")
activate(call, call.ring(klaus, timeOfCall))
simulate(until=maxTime)
```

```
 0.0000 Klaus: Here I am
 9.0000 Ringgg!
 9.0000 Klaus: Excuse me
 9.0000 Klaus: Hello! I'll call back
12.0000 Klaus: Sorry, where were we?
23.0000 Klaus: I must leave
```

As this has no random numbers the results are reasonably clear: the interrupting call occurs at 9.0. It takes `klaus` 3 minutes to listen to the message and he resumes the conversation with the bank manager at 12.0. His total time of conversation is 9.0 + 11.0 = 20.0 minutes as it would have been if the interrupt had not occurred.

3.9.2 `waituntil` the Bank door opens

Customers arrive at random, some of them getting to the bank before the door is opened by a doorman. They wait for the door to be opened and then rush in and queue to be served.

This model uses the `waituntil` yield command. In the program listing the door is initially closed (label #1) and a function to test if it is open is defined at label #2.

The `Doorman` class is defined starting at label #3 and the single `doorman` is created and activated at at labels #7 and #8. The doorman waits for an average 10 minutes (label #4) and then opens the door.

The `Customer` class is defined at label #5 and a new customer prints out `Here I am` on arrival. If the door is still closed, he adds `but the door is shut` and settles down to wait (label #6), using the `yield waituntil` command. When the door is opened by the doorman the `dooropen` state is changed and the customer (and all others waiting for the door) proceed. A customer arriving when the door is open will not be delayed.

```
"""bank14: *waituntil* the Bank door opens"""
from SimPy.Simulation import *
from random import expovariate, seed

# Model components ------------------------
door = 'Shut'  # 1


def dooropen():  # 2
    return door == 'Open'
```

```
class Doorman(Process):  # 3
    """ Doorman opens the door"""

    def openthedoor(self):
        """ He will open the door when he arrives"""
        global door
        yield hold, self, expovariate(1.0 / 10.0)  # 4
        door = 'Open'
        print("%7.4f Doorman: Ladies and "
              "Gentlemen! You may all enter." % (now()))


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):  # 5
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank=10):
        arrive = now()
        if dooropen():
            msg = ' and the door is open.'
        else:
            msg = ' but the door is shut.'
        print("%7.4f %s: Here I am%s" % (now(), self.name, msg))

        yield waituntil, self, dooropen  # 6

        print("%7.4f %s: I can  go in!" % (now(), self.name))
        wait = now() - arrive
        print("%7.4f %s: Waited %6.3f" % (now(), self.name, wait))

        yield request, self, counter
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        print("%7.4f %s: Finished    " % (now(), self.name))

# Experiment data -------------------------


maxTime = 2000.0   # minutes
counter = Resource(1, name="Clerk")

# Model  ----------------------------------


def model(SEED=393939):
    seed(SEED)
```

```
    initialize()
    door = 'Shut'
    doorman = Doorman()  # 7
    activate(doorman, doorman.openthedoor())  # 8
    source = Source()
    activate(source,
             source.generate(number=5, rate=0.1), at=0.0)
    simulate(until=400.0)


# Experiment  ----------------------------------
model()
```

The output from a run for this programs shows how the first customer has to wait until the door is opened.

```
 0.0000 Customer00: Here I am but the door is shut.
 1.1489 Doorman: Ladies and Gentlemen! You may all enter.
 1.1489 Customer00: I can  go in!
 1.1489 Customer00: Waited  1.149
 6.5691 Customer00: Finished
 8.3438 Customer01: Here I am and the door is open.
 8.3438 Customer01: I can  go in!
 8.3438 Customer01: Waited  0.000
15.5704 Customer02: Here I am and the door is open.
15.5704 Customer02: I can  go in!
15.5704 Customer02: Waited  0.000
21.2664 Customer03: Here I am and the door is open.
21.2664 Customer03: I can  go in!
21.2664 Customer03: Waited  0.000
21.9473 Customer04: Here I am and the door is open.
21.9473 Customer04: I can  go in!
21.9473 Customer04: Waited  0.000
27.6401 Customer01: Finished
56.5248 Customer02: Finished
57.3640 Customer03: Finished
77.3587 Customer04: Finished
```

3.9.3 Wait for the doorman to give a signal: `waitevent`

Customers arrive at random, some of them getting to the bank before the door is open. This is controlled by an automatic machine called the doorman which opens the door only at intervals of 30 minutes (it is a very secure bank). The customers wait for the door to be opened and all those waiting enter and proceed to the counter. The door is closed behind them.

This model uses the `yield waitevent` command which requires a `SimEvent` to be defined (label #1). The `Doorman` class is defined at label #2 and the `doorman` is created and activated at labels #7 and #8. The doorman waits for a fixed time (label #3) and then tells the customers that the door is open. This is achieved on label #4 by signalling the `dooropen` event.

The `Customer` class is defined at label #5 and in its PEM, when a customer arrives, he prints out `Here I am`. If the door is still closed, he adds *"but the door is shut"* and settles down to wait for the door to be opened using the `yield waitevent` command (label #6). When the door is opened by the doorman (that is, he sends the `dooropen.signal()` the customer and any others waiting may proceed.

```
""" bank13: Wait for the doorman to give a signal: *waitevent*"""
from SimPy.Simulation import *
```

```
from random import *

# Model components ------------------------

dooropen = SimEvent("Door Open")  # 1


class Doorman(Process):  # 2
    """ Doorman opens the door"""

    def openthedoor(self):
        """ He will opens the door at fixed intervals"""
        for i in range(5):
            yield hold, self, 30.0  # 3
            dooropen.signal()  # 4
            print("%7.4f You may enter" % (now()))


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):  # 5
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank=10):
        arrive = now()

        if dooropen.occurred:
            msg = '.'
        else:
            msg = ' but the door is shut.'
        print("%7.4f %s: Here I am%s" % (now(), self.name, msg))
        yield waitevent, self, dooropen

        print("%7.4f %s: The door is open!" % (now(), self.name))

        wait = now() - arrive
        print("%7.4f %s: Waited %6.3f" % (now(), self.name, wait))

        yield request, self, counter
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        print("%7.4f %s: Finished    " % (now(), self.name))

# Experiment data -------------------------


maxTime = 400.0  # minutes
```

```
counter = Resource(1, name="Clerk")

# Model  ----------------------------------


def model(SEED=232323):
    seed(SEED)

    initialize()
    doorman = Doorman()  # 7
    activate(doorman, doorman.openthedoor())  # 8
    source = Source()
    activate(source,
             source.generate(number=5, rate=0.1), at=0.0)
    simulate(until=maxTime)


# Experiment  ----------------------------------

model()
```

An output run for this programs shows how the first three customers have to wait until the door is opened.

```
 0.0000 Customer00: Here I am but the door is shut.
13.6767 Customer01: Here I am but the door is shut.
13.9068 Customer02: Here I am but the door is shut.
30.0000 You may enter
30.0000 Customer02: The door is open!
30.0000 Customer02: Waited 16.093
30.0000 Customer01: The door is open!
30.0000 Customer01: Waited 16.323
30.0000 Customer00: The door is open!
30.0000 Customer00: Waited 30.000
34.0411 Customer03: Here I am but the door is shut.
40.8095 Customer04: Here I am but the door is shut.
55.4721 Customer02: Finished
57.2363 Customer01: Finished
60.0000 You may enter
60.0000 Customer04: The door is open!
60.0000 Customer04: Waited 19.190
60.0000 Customer03: The door is open!
60.0000 Customer03: Waited 25.959
77.0409 Customer00: Finished
90.0000 You may enter
104.8327 Customer04: Finished
118.4142 Customer03: Finished
120.0000 You may enter
150.0000 You may enter
```

3.10 Monitors

Monitors (and Tallys) are used to track and record values in a simulation. They store a list of [time,value] pairs, one pair being added whenever the `observe` method is called. A particularly useful characteristic is that they continue to exist after the simulation has been completed. Thus further analysis of the results can be carried out.

Monitors have a set of simple statistical methods such as `mean` and `var` to calculate the average and variance of the observed values – useful in estimating the mean delay, for example.

They also have the `timeAverage` method that calculates the time-weighted average of the recorded values. It determines the total area under the time~value graph and divides by the total time. This is useful for estimating the average number of customers in the bank, for example. There is an *important caveat* in using this method. To estimate the correct time average you must certainly `observe` the value (say the number of customers in the system) whenever it changes (as well as at any other time you wish) but, and this is important, observing the *new* value. The *old* value was recorded earlier. In practice this means that if we wish to observe a changing value, `n`, using the Monitor, `Mon`, we must keep to the the following pattern:

```
n = n+1
Mon.observe(n,now())
```

Thus you make the change (not only increases) and *then* observe the new value. Of course the simulation time `now()` has not changed between the two statements.

3.10.1 Plotting a Histogram of Monitor results

A Monitor can construct a histogram from its data using the `histogram` method. In this model we monitor the time in the system for the customers. This is calculated for each customer at label #2, using the arrival time saved at label #1. We create the Monitor, `Mon`, at label #4 and the times are `observed` at label #3.

The histogram is constructed from the Monitor, after the simulation has finished, at label #5. The SimPy SimPlot package allows simple plotting of results from simulations. Here we use the SimPlot `plotHistogram` method. The plotting routines appear between labels #6 and #7. The `plotHistogram` call is in label #7.

```
"""bank17: Plotting a Histogram of Monitor results"""
from SimPy.Simulation import *
from SimPy.SimPlot import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank):
        arrive = now()  # 1
```

```
        # print("%8.4f %s: Arrived      " % (now(), self.name))

        yield request, self, counter
        # print("%8.4f %s: Got counter " % (now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        # print("%8.4f %s: Finished    " % (now(), self.name))
        t = now() - arrive  # 2
        Mon.observe(t)  # 3

# Experiment data ------------------------


maxTime = 400.0   # minutes
counter = Resource(1, name="Clerk")
Mon = Monitor('Time in the Bank')  # 4
N = 0

# Model  ---------------------------------


def model(SEED=393939):
    seed(SEED)

    initialize()
    source = Source()
    activate(source,
             source.generate(number=20, rate=0.1), at=0.0)
    simulate(until=maxTime)


# Experiment  ----------------------------------
model()
Histo = Mon.histogram(low=0.0, high=200.0, nbins=20)  # 5

plt = SimPlot()  # 6
plt.plotHistogram(Histo, xlab='Time (min)',  # 7
                  title="Time in the Bank",
                  color="red", width=2)
plt.mainloop()  # 8
```

3.10.2 Plotting from Resource Monitors

Like all Monitors, `waitMon` and `actMon` in a monitored Resource contain information that enables us to graph the output. Alternative plotting packages can be used; here we use the simple `SimPy.SimPlot` package just to graph the number of customers waiting for the counter. The program is a simple modification of the one that uses a monitored Resource.

The SimPlot package is imported at #3. No major changes are made to the main part of the program except that I commented out the print statements. The changes occur in the `model` routine from #3 to #5.The simulation now generates and processes 20 customers (#4). `model` does not return a value but the Monitors of the `counter` Resource still exist when the simulation has terminated.

The additional plotting actions take place from #6 to #9. Lines #7 and #8 construct a step plot and graphs the number in the waiting queue as a function of time. `waitMon` is primarily a list of *[time,value]* pairs which the `plotStep`

method of the SimPlot object, `plt` uses without change. On running the program the graph is plotted; the user has to terminate the plotting `mainloop` on the screen.

```
"""bank16: Plotting from  Resource Monitors"""
from SimPy.Simulation import *
from SimPy.SimPlot import *  # 1
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i))
            activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank):
        arrive = now()
        # print("%8.4f %s: Arrived     " % (now(), self.name))

        yield request, self, counter
        # print("%8.4f %s: Got counter " % (now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        # print("%8.4f %s: Finished    " % (now(), self.name))

# Experiment data -------------------------


maxTime = 400.0   # minutes
counter = Resource(1, name="Clerk", monitored=True)

# Model -----------------------------------


def model(SEED=393939):  # 3
    seed(SEED)
    initialize()
    source = Source()
    activate(source,  # 4
             source.generate(number=20, rate=0.1), at=0.0)
    simulate(until=maxTime)  # 5

# Experiment ------------------------------------


model()

plt = SimPlot()  # 6
```

```
plt.plotStep(counter.waitMon,  # 7
             color="red", width=2)  # 8
plt.mainloop()  # 9
```

3.11 Acknowledgements

I thank Klaus Muller, Bob Helmbold, Mukhlis Matti and the other developers and users of SimPy for improving this document by sending their comments. I would be grateful for any further corrections or suggestions. Please send them to: *vignaux* at *users.sourceforge.net*.

3.12 References

- Python website: https://www.python.org
- SimPy homepage: https://github.com/SimPyClassic/SimPyClassic
- The Bank:

3.13 The Bank (Object Oriented version)

3.13.1 Introduction

SimPy is used to develop a simple simulation of a bank with a number of tellers. This Python package provides *Processes* to model active components such as messages, customers, trucks, and planes. It has three classes to model facilities where congestion might occur: *Resources* for ordinary queues, *Levels* for the supply of quantities of material, and *Stores* for collections of individual items. Only examples of *Resources* are described here. It also provides *Monitors* and *Tallys* to record data like queue lengths and delay times and to calculate simple averages. It uses the standard Python random package to generate random numbers.

Starting with SimPy 2.0 an object-oriented programmer's interface was added to the package and it is this version that is described here. It is quite compatible with the procedural approach. The object-oriented interface, however, can support the process of developing and extending a simulation model better than the procedural approach.

SimPy can be obtained from: https://github.com/SimPyClassic/SimPyClassic. This tutorial is best read with the SimPy Manual and CheatsheetOO at your side for reference.

Before attempting to use SimPy you should be familiar with the Python language. In particular you should be able to use *classes*. Python is free and available for most machine types. You can find out more about it at the Python web site. SimPy is compatible with Python version 2.3 and later.

3.13.2 A single Customer

In this tutorial we model a simple bank with customers arriving at random. We develop the model step-by-step, starting out simply, and producing a running program at each stage. The programs we develop are available without line numbers and ready to go, in the `bankprograms_OO` directory. Please copy them, run them and improve them - and in the tradition of open-source software suggest your modifications to the SimPy users list. Object-orented versions of all the models are included in the same directory.

A simulation should always be developed to answer a specific question; in these models we investigate how changing the number of bank servers or tellers might affect the waiting time for customers.

A Customer arriving at a fixed time

We first model a single customer who arrives at the bank for a visit, looks around at the decor for a time and then leaves. There is no queueing. First we will assume his arrival time and the time he spends in the bank are fixed.

Examine the following listing which is a complete runnable Python script, except for the line numbers. We use comments to divide the script up into sections. This makes for clarity later when the programs get more complicated. Line 1 is a normal Python documentation string; line 2 makes available the SimPy constructs needed for this model: the `Simulation` class, the `Process` class, and the `hold` verb.

We define a `Customer` class derived from the SimPy `Process` class. We create a `Customer` object, `c` who arrives at the bank at simulation time `5.0` and leaves after a fixed time of `10.0` minutes. The `Customer` class definition, lines 5-12, defines our customer class and has the required generator method (called `visit`) (line 9) having a `yield` statement (line 11)). Such a method is called a Process Execution Method (PEM) in SimPy.

The customer's `visit` PEM, lines 9-12, models his activities. When he arrives (it will turn out to be a 'he' in this model), he will print out the simulation time, `self.sim.now()`, and his name (line 10). `self.sim` is a reference to the `BankModel` simulation object where this customer exists. Every `Process` instance is linked to the simulation in which it is created by assigning to its `sim` parameter when it is created (see line 19). The method `now()` can be used at any time in the simulation to find the current simulation time though it cannot be changed by the programmer. The customer's name will be set when the customer is created later in the script (line 19).

He then stays in the bank for a fixed simulation time `timeInBank` (line 11). This is achieved by the `yield hold, self,timeInBank` statement. This is the first of the special simulation commands that `SimPy` offers.

After a simulation time of `timeInBank`, the program's execution returns to the line after the `yield` statement, line 12. The customer then prints out the current simulation time and his name. This completes the declaration of the `Customer` class.

Though we do not do it here, it is also possible to define an `__init__()` method for a `Process` if you need to give the customer any attributes. Bear in mind that such an `__init__` method must first call `Process.__init__(self)` and can then initialize any instance variables needed.

Lines 6 to 21 define a class `BankModel`, composing a model of a bank from the `Simulation` class, a `Customer` class and the global experiment data. The definition `class BankModel(Simulation)` gives an instance of a `BankModel` all the attributes of class `Simulation`. (In OO terms, `BankModel` *inherits* from `Simulation`.) Any instance of `BankModel` *is* a `Simulation` instance. This gives a `BankModel` its own event list and thus its own time axis. Also, it allows a `BankModel` instance to activate processes and to start the execution of a simulation on its time axis.

Lines 17 to 21 define a `run` method; when called, it results in the execution of a `BankModel` instance, i.e. the performance of a simulation experiment. Line 18 initializes this simulation, i.e. it creates a new event list. L.19 creates a `Customer` object. The parameter assignment `sim = self` ties the customer instance to this and only this simulation. The customer does not exist outside this simulation. L.20 activates the customer's `visit` process (PEM). Finally the call of `simulate(until=maxTime)` in line 24 starts the simulation. It will run until the simulation time is `maxTime` unless stopped beforehand either by the `stopSimulation()` command or by running out of events to execute (as will happen here). `maxTime` was set to `100.0` in line 25.

Note: If model classes like the''BankModel'' are to be given any other attributes, they must have an `__init__` method in which these attributes are assigned with the syntax `self.attrib1 = . . ..` Such an `__init__` method must first call `Simulation.__init__(self)` to also initialize the `Simulation` class from which the model inherits.

The simulation model is executed by line 32. `BankModel()` constructs the model, and `.run()` executes it. This is just a short form of:

```
bM = BankModel()
bM.run()
```

```
""" bank01_OO: The single non-random Customer """
from SimPy.Simulation import Simulation, Process, hold  # 1
# Model components ------------------------------


class Customer(Process):  # 2
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):
        print("%2.1f %s  Here I am" %
              (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%2.1f %s  I must leave" %
              (self.sim.now(), self.name))

# Model -----------------------------------------


class BankModel(Simulation):  # 3
    def run(self):
        """ An independent simulation object """
        c = Customer(name="Klaus", sim=self)  # 4
        self.activate(c, c.visit(timeInBank), at=tArrival)
        self.simulate(until=maxTime)  # 5


# Experiment data -------------------------------
maxTime = 100.0     # minutes                          #6
timeInBank = 10.0   # minutes
tArrival = 5.0      # minutes

# Experiment ------------------------------------
mymodel = BankModel()  # 7
mymodel.run()  # 8
```

The short trace printed out by the `print` statements shows the result. The program finishes at simulation time `15.0` because there are no further events to be executed. At the end of the `visit` routine, the customer has no more actions and no other objects or customers are active.

```
5.0 Klaus  Here I am
15.0 Klaus  I must leave
```

A Customer arriving at random

Now we extend the model to allow our customer to arrive at a random simulated time though we will keep the time in the bank at 10.0, as before.

The change occurs in line 3 of the program and in lines 19, 21, 23, 31 and 35. In line 3 we import from the standard Python `random` module to give us `expovariate` to generate the random time of arrival. We also import the `seed` function to initialize the random number stream to allow control of the random numbers. The `run` method is given a parameter `aseed` for the initial seed (line 19) .In line 31 we provide an initial seed of `99999`. An exponential random variate, `t`, is generated in line 23. Note that the Python Random module's `expovariate` function uses the

rate (here, `1.0/tMeanArrival`) as the argument. The generated random variate, `t`, is used in line 24 as the `at` argument to the `activate` call. `tMeanArrival` is assigned a value of `5.0` minutes at line 31.

In line 35, the `BankModel` entity is generated and its `run` function called with parameter assignment `aseed=seedVal`.

```
""" bank05_OO: The single Random Customer """
from SimPy.Simulation import Simulation, Process, hold
from random import expovariate, seed

# Model components ------------------------


class Customer(Process):
    """ Customer arrives at a random time,
        looks around  and then leaves """

    def visit(self, timeInBank):
        print("%f %s Here I am" % (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%f %s I must leave" % (self.sim.now(), self.name))

# Model -----------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        c = Customer(name="Klaus", sim=self)
        t = expovariate(1.0 / tMeanArrival)
        self.activate(c, c.visit(timeInBank), at=t)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 100.0     # minutes
timeInBank = 10.0   # minutes
tMeanArrival = 5.0  # minutes
seedVal = 99999

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

The result is shown below. The customer now arrives at time 10.5809. Changing the seed value would change that time.

```
0.641954 Klaus Here I am
10.641954 Klaus I must leave
```

The display looks pretty untidy. In the next example I will try and make it tidier.

3.13.3 More Customers

Our simulation does little so far. To consider a simulation with several customers we return to the simple deterministic model and add more `Customers`.

The program is almost as easy as the first example (*A Customer arriving at a fixed time*). The main change is in lines 19-24 where we create, name, and activate three customers. We also increase the maximum simulation time to `400` (line 29 and referred to in line 25). Observe that we need only one definition of the `Customer` class and create several objects of that class. These will act quite independently in this model.

Each customer stays for a different `timeInBank` so, instead of setting a common value for this we set it for each customer. The customers are started at different times (using `at=`). `Tony's` activation time occurs before `Klaus's`, so `Tony` will arrive first even though his activation statement appears later in the script.

As promised, the print statements have been changed to use Python string formatting (lines 10 and 12). The statements look complicated but the output is much nicer.

```
""" bank02_OO: More Customers """
from SimPy.Simulation import Simulation, Process, hold


# Model components ------------------------
class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank=0):
        print("%7.4f %s: Here I am" % (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (self.sim.now(), self.name))


# Model -----------------------------------
class BankModel(Simulation):
    def run(self):
        """ PEM """
        c1 = Customer(name="Klaus", sim=self)
        self.activate(c1, c1.visit(timeInBank=10.0), at=5.0)
        c2 = Customer(name="Tony", sim=self)
        self.activate(c2, c2.visit(timeInBank=7.0), at=2.0)
        c3 = Customer(name="Evelyn", sim=self)
        self.activate(c3, c3.visit(timeInBank=20.0), at=12.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 400.0  # minutes

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run()
```

The trace produced by the program is shown below. Again the simulation finishes before the `400.0` specified in the `simulate` call.

```
 2.0000 Tony: Here I am
 5.0000 Klaus: Here I am
 9.0000 Tony: I must leave
```

```
12.0000 Evelyn: Here I am
15.0000 Klaus: I must leave
32.0000 Evelyn: I must leave
```

Many Customers

Another change will allow us to have more customers. As it is tedious to give a specially chosen name to each one, we will call them `Customer00, Customer01,...` and use a separate `Source` class to create and activate them. To make things clearer we do not use the random numbers in this model.

The following listing shows the new program. Lines 6-13 define a `Source` class. Its PEM, here called `generate`, is defined in lines 9-13. This PEM has a couple of arguments: the `number` of customers to be generated and the Time Between Arrivals, `TBA`. It consists of a loop that creates a stream of numbered `Customers` from `0` to `(number-1)`, inclusive. We create a customer and give it a name in line 11. The parameter assignment `sim = self.sim` ties the customers to the `BankModel` to which the `Source` belongs. The customer is then activated at the current simulation time (the final argument of the `activate` statement is missing so that the default value of `self.sim.now()`, the current simulation time for the instance of `BankModel`, is used as the time; here, it is `0.0`). We also specify how long the customer is to stay in the bank. To keep it simple, all customers stay exactly `12` minutes. After each new customer is activated, the `Source` holds for a fixed time (`yield hold,self,TBA`) before creating the next one (line 13).

`class BankModel(Simulation)` (line 24) provides a `run` method which executes this model consisting of a customer source and the global data. As `BankModel` inherits from `Simulation`, it has its own event list which gets initialized as empty in line 26.

A `Source`, `s`, is created in line 27 and activated at line 28 where the number of customers to be generated is set to `maxNumber = 5` and the interval between customers to `ARRint = 10.0`. The parameter assignment `sim = self` links the `Source` process to this `BankModel` instance. Once started at time `0.0`, `s` creates customers at intervals and each customer then operates independently of the others.

In line 40, a `BankModel` object is created and its `run` method executed:

```
""" bank03_OO: Many non-random Customers """
from SimPy.Simulation import Simulation, Process, hold


# Model components ------------------------
class Source(Process):
    """ Source generates customers regularly """

    def generate(self, number, TBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            yield hold, self, TBA


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank):
        print("%7.4f %s: Here I am" % (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (self.sim.now(), self.name))
```

```
# Model ----------------------------------
class BankModel(Simulation):
    def run(self):
        """ PEM """
        s = Source(sim=self)
        self.activate(s, s.generate(number=maxNumber,
                                    TBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0  # minutes
ARRint = 10.0    # time between arrivals, minutes

# Experiment -----------------------------

mymodel = BankModel()
mymodel.run()
```

The output is:

```
 0.0000 Customer00: Here I am
10.0000 Customer01: Here I am
12.0000 Customer00: I must leave
20.0000 Customer02: Here I am
22.0000 Customer01: I must leave
30.0000 Customer03: Here I am
32.0000 Customer02: I must leave
40.0000 Customer04: Here I am
42.0000 Customer03: I must leave
52.0000 Customer04: I must leave
```

Many Random Customers

We now extend this model to allow arrivals at random. In simulation this is usually interpreted as meaning that the times between customer arrivals are distributed as exponential random variates. There is little change in our program, we use a `Source` object, as before.

The exponential random variate is generated in line 14 with `meanTBA` as the mean Time Between Arrivals and used in line 15. Note that this parameter is not exactly intuitive. As already mentioned, the Python `expovariate` method uses the *rate* of arrivals as the parameter not the average interval between them. The exponential delay between two arrivals gives pseudo-random arrivals. In this model the first customer arrives at time `0.0`.

The `seed` method is called to initialize the random number stream in the `run` routine of `BankModel` (line 30). It uses the value provided by parameter `aseed`. It is possible to leave this call out but if we wish to do serious comparisons of systems, we must have control over the random variates and therefore control over the seeds. Then we can run identical models with different seeds or different models with identical seeds. We provide the seeds as control parameters of the run. Here a seed is assigned in line 41 but it is clear it could have been read in or manually entered on an input form.

The `BankModel` is generated in line 45 and its `run` method called with the seed value as parameter.

```
""" bank06: Many Random Customers """
from SimPy.Simulation import Simulation, Process, hold
from random import expovariate, seed
```

```
# Model components ------------------------
class Source(Process):
    """ Source generates customers at random """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, looks round and leaves """

    def visit(self, timeInBank=0):
        print("%7.4f %s: Here I am" % (self.sim.now(), self.name))
        yield hold, self, timeInBank
        print("%7.4f %s: I must leave" % (self.sim.now(), self.name))


# Model -----------------------------------
class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        s = Source(name='Source', sim=self)
        self.activate(s, s.generate(number=maxNumber,
                                    meanTBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0  # minutes
ARRint = 10.0    # mean arrival interval, minutes
seedVal = 99999

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

This generates the following output:

```
 0.0000 Customer00: Here I am
 1.2839 Customer01: Here I am
 4.9842 Customer02: Here I am
12.0000 Customer00: I must leave
13.2839 Customer01: I must leave
16.9842 Customer02: I must leave
35.5432 Customer03: Here I am
47.5432 Customer03: I must leave
48.9918 Customer04: Here I am
60.9918 Customer04: I must leave
```

3.13.4 A Service counter

So far, the model bank has been more like an art gallery, the customers entering, looking around, and leaving. Now they are going to require service from the bank clerk. We extend the model to include a service counter which will be modelled as an object of SimPy's `Resource` class with a single resource unit. The actions of a `Resource` are simple: a customer `requests` a unit of the resource (a clerk). If one is free he gets service (and removes the unit, i.e., makes it busy). If there is no free clerk the customer joins the queue (managed by the resource object) until it is his turn to be served. As each customer completes service and `releases` the unit, the clerk automatically starts serving the next in line. This is done by reactivating that customer's process where it had been blocked.

One Service counter

As this model is built with the `Resource` class from `SimPy.Simulation`, it and the related `request` and `release` verbs are mported, in addition to the imports made in the previous programs (line 2).

The service counter is created as a `Resource` attribute `self.k` of the `BankModel` (line 39). The resource exists in the `BankModel`, and this is indicated by the parameter assignment `sim = self`. The `Source` PEM `generate` can access this attribute by `self.sim.k`, its `BankModel`'s resource attribute (line 14).

The actions involving the `Counter` referred to by the parameter `res` in the customer's PEM are:

- the `yield request` statement in line 25. If the server is free then the customer can start service immediately and the code moves on to line 26. If the server is busy, the customer is automatically queued by the Resource. When it eventually comes available the PEM moves on to line 26.
- the `yield hold` statement in line 28 where the operation of the service counter is modelled. Here the service time is a fixed `timeInBank`. During this period the customer is being served and the resource (the counter) is busy.
- the `yield release` statement in line 29. The current customer completes service and the service counter becomes available for any remaining customers in the queue.

Observe that the service counter is used with the pattern (`yield request..; yield hold..; yield release..`).

To show the effect of the service counter on the activities of the customers, I have added line 22 to record when the customer arrived and line 26 to record the time between arrival in the bank and starting service. Line 26 is *after* the `yield request` command and will be reached only when the request is satisfied. It is *before* the `yield hold` that corresponds to the start of service. The variable `wait` will record how long the customer waited and will be 0 if he received service at once. This technique of saving the arrival time in a variable is common. So the `print` statement also prints out how long the customer waited in the bank before starting service.

```
""" bank07_OO: One Counter,random arrivals """
from SimPy.Simulation import (Simulation, Process, hold, Resource, request,
                              release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0,
                                         res=self.sim.k))
```

```
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0, res=None):
        arrive = self.sim.now()       # arrival time
        print("%8.3f %s: Here I am" % (self.sim.now(), self.name))

        yield request, self, res
        wait = self.sim.now() - arrive  # waiting time
        print("%8.3f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        yield hold, self, timeInBank
        yield release, self, res

        print("%8.3f %s: Finished" % (self.sim.now(), self.name))

# Model ------------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.k = Resource(name="Counter", unitName="Clerk", sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, meanTBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data --------------------------


maxNumber = 5
maxTime = 400.0  # minutes
ARRint = 10.0    # mean, minutes
seedVal = 99999

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

Examining the trace we see that the first two customers get instant service but the others have to wait. We still only have five customers (line 44) so we cannot draw general conclusions.

```
   0.000 Customer00: Here I am
   0.000 Customer00: Waited  0.000
   1.284 Customer01: Here I am
   4.984 Customer02: Here I am
  12.000 Customer00: Finished
  12.000 Customer01: Waited 10.716
  24.000 Customer01: Finished
  24.000 Customer02: Waited 19.016
  35.543 Customer03: Here I am
  36.000 Customer02: Finished
  36.000 Customer03: Waited  0.457
```

```
 48.000 Customer03: Finished
 48.992 Customer04: Here I am
 48.992 Customer04: Waited  0.000
 60.992 Customer04: Finished
```

A server with a random service time

This is a simple change to the model in that we retain the single service counter but make the customer service time a random variable. As is traditional in the study of simple queues we first assume an exponential service time and set the mean to `timeInBank`.

The service time random variable, `tib`, is generated in line 26 and used in line 27. The argument to be used in the call of `expovariate` is not the mean of the distribution, `timeInBank`, but is the rate `1.0/timeInBank`.

We have put together the exeriment data by defining a number of appropriate variables and giving them values. These are in lines 44 to 48.

```
""" bank08_OO: A counter with a random service time """
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import expovariate, seed


# Model components ------------------------

class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(b=self.sim.k))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = self.sim.now()
        print("%8.4f %s: Here I am     " % (self.sim.now(), self.name))
        yield request, self, b
        wait = self.sim.now() - arrive
        print("%8.4f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b
        print("%8.4f %s: Finished      " % (self.sim.now(), self.name))

# Model -----------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
```

```
        self.k = Resource(name="Counter", unitName="Clerk", sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, meanTBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0    # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
seedVal = 99999

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

And the output:

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Waited  0.000
  1.2839 Customer01: Here I am
  4.4403 Customer00: Finished
  4.4403 Customer01: Waited  3.156
 20.5786 Customer01: Finished
 31.8430 Customer02: Here I am
 31.8430 Customer02: Waited  0.000
 34.5594 Customer02: Finished
 36.2308 Customer03: Here I am
 36.2308 Customer03: Waited  0.000
 41.4313 Customer04: Here I am
 67.1315 Customer03: Finished
 67.1315 Customer04: Waited 25.700
 87.9241 Customer04: Finished
```

This model with random arrivals and exponential service times is an example of an M/M/1 queue and could rather easily be solved analytically to calculate the steady-state mean waiting time and other operating characteristics. (But not so easily solved for its transient behavior.)

3.13.5 Several Service Counters

When we introduce several counters we must decide on a queue discipline. Are customers going to make one queue or are they going to form separate queues in front of each counter? Then there are complications - will they be allowed to switch lines (jockey)? We first consider a single queue with several counters and later consider separate isolated queues. We will not look at jockeying.

Several Counters but a Single Queue

Here we model a bank whose customers arrive randomly and are to be served at a group of counters, taking a random time for service, where we assume that waiting customers form a single first-in first-out queue.

The *only* difference between this model and the single-server model is in line 37. We have provided two counters by increasing the capacity of the `counter` resource to 2. This value is set in line 50 (`Nc = 2`). These *units* of the

resource correspond to the two counters. Because both clerks cannot be called `Karen`, we have used a general name of `Clerk` as resource unit.

```
""" bank09_OO: Several Counters but a Single Queue """
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(b=self.sim.k))
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = self.sim.now()
        print("%8.4f %s: Here I am     " % (self.sim.now(), self.name))
        yield request, self, b
        wait = self.sim.now() - arrive
        print("%8.4f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b
        print("%8.4f %s: Finished      " % (self.sim.now(), self.name))

# Model -----------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.k = Resource(capacity=Nc, name="Counter", unitName="Clerk",
                          sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, meanTBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxNumber = 5       # of customers
maxTime = 400.0     # minutes
timeInBank = 12.0   # mean, minutes
ARRint = 10.0       # mean, minutes
Nc = 2              # of clerks/counters
seedVal = 99999
```

```
# Experiment ------------------------------
mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

The waiting times in this model are much shorter than those for the single service counter. For example, the waiting time for `Customer02` has been reduced from `51.213` to `12.581` minutes. Again we have too few customers processed to draw general conclusions.

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Waited  0.000
  1.2839 Customer01: Here I am
  1.2839 Customer01: Waited  0.000
  4.4403 Customer00: Finished
 17.4222 Customer01: Finished
 31.8430 Customer02: Here I am
 31.8430 Customer02: Waited  0.000
 34.5594 Customer02: Finished
 36.2308 Customer03: Here I am
 36.2308 Customer03: Waited  0.000
 41.4313 Customer04: Here I am
 41.4313 Customer04: Waited  0.000
 62.2239 Customer04: Finished
 67.1315 Customer03: Finished
```

Several Counters with individual queues

Each counter is now assumed to have its own queue. The programming is more complicated because the customer has to decide which queue to join. The obvious technique is to make each counter a separate resource and it is useful to make a list of resource objects (line 56).

In practice, a customer will join the shortest queue. So we define the Python function, `NoInSystem(R)` (lines 17-19) which returns the sum of the number waiting and the number being served for a particular counter, `R`. This function is used in line 28 to list the numbers at each counter. It is then easy to find which counter the arriving customer should join. We have also modified the trace printout, line 29 to display the state of the system when the customer arrives. We choose the shortest queue in lines 30-33 (the variable `choice`).

The rest of the program is the same as before.

```
""" bank10_OO: Several Counters with individual queues"""
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i,), sim=self.sim)
            self.sim.activate(c, c.visit(counters=self.sim.kk))
            t = expovariate(1.0 / interval)
            yield hold, self, t
```

```
def NoInSystem(R):
    """ Total number of customers in the resource R"""
    return (len(R.waitQ) + len(R.activeQ))


class Customer(Process):
    """ Customer arrives, chooses the shortest queue
        is served and leaves
    """

    def visit(self, counters):
        arrive = self.sim.now()
        Qlength = [NoInSystem(counters[i]) for i in range(Nc)]
        print("%7.4f %s: Here I am. %s" % (self.sim.now(), self.name, Qlength))
        for i in range(Nc):
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                choice = i  # the chosen queue number
                break

        yield request, self, counters[choice]
        wait = self.sim.now() - arrive
        print("%7.4f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counters[choice]

        print("%7.4f %s: Finished" % (self.sim.now(), self.name))


# Model --------------------------------

class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.kk = [Resource(name="Clerk0", sim=self),
                   Resource(name="Clerk1", sim=self)]
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, interval=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxNumber = 5
maxTime = 400.0    # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
Nc = 2             # number of counters
seedVal = 787878

# Experiment ------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

The results show how the customers choose the counter with the smallest number. Unlucky `Customer02` who joins

the wrong queue has to wait until Customer00 finishes at time 55.067. There are, however, too few arrivals in these runs, limited as they are to five customers, to draw any general conclusions about the relative efficiencies of the two systems.

```
 0.0000 Customer00: Here I am. [0, 0]
 0.0000 Customer00: Waited  0.000
 9.7519 Customer00: Finished
12.0829 Customer01: Here I am. [0, 0]
12.0829 Customer01: Waited  0.000
25.9167 Customer02: Here I am. [1, 0]
25.9167 Customer02: Waited  0.000
38.2349 Customer03: Here I am. [1, 1]
40.4032 Customer04: Here I am. [2, 1]
43.0677 Customer02: Finished
43.0677 Customer04: Waited  2.664
44.0242 Customer01: Finished
44.0242 Customer03: Waited  5.789
60.1271 Customer03: Finished
70.2500 Customer04: Finished
```

3.13.6 Monitors and Gathering Statistics

The traces of output that have been displayed so far are valuable for checking that the simulation is operating correctly but would become too much if we simulate a whole day. We do need to get results from our simulation to answer the original questions. What, then, is the best way to summarize the results?

One way is to analyze the traces elsewhere, piping the trace output, or a modified version of it, into a *real* statistical program such as *R* for statistical analysis, or into a file for later examination by a spreadsheet. We do not have space to examine this thoroughly here. Another way of presenting the results is to provide graphical output.

SimPy offers an easy way to gather a few simple statistics such as averages: the Monitor and Tally classes. The Monitor records the values of chosen variables as time series. (but see the comments in *Final Remarks*).

The Bank with a Monitor

We now demonstrate a Monitor that records the average waiting times for our customers. We return to the system with random arrivals, random service times and a single queue and remove the old trace statements. In practice, we would make the printouts controlled by a variable, say, TRACE which is set in the experimental data (or read in as a program option - but that is a different story). This would aid in debugging and would not complicate the data analysis. We will run the simulations for many more arrivals.

In addition to the imports in the programs shown before, we now have to import the Monitor class (line 2).

A Monitor, wM, is created in line 37. We make the monitor an attribute of the BankModel by the assignment to self.wM. The monitor observes the waiting time mentioned in line 25. As the monitor is an attribute of the BankModel to which the customer belongs, self.sim.wM can refere to it. We run maxNumber = 50 customers (in the call of generate in line 39) and have increased maxTime to 1000.0 minutes.

```
""" bank11: The bank with a Monitor"""
from SimPy.Simulation import Simulation, Process, Resource, Monitor, hold,\
    request, release
from random import expovariate, seed

# Model components ------------------------
```

```
class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(b=self.sim.k))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = self.sim.now()
        yield request, self, b
        wait = self.sim.now() - arrive
        self.sim.wM.observe(wait)
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, b

# Model --------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.k = Resource(capacity=Nc, name="Clerk", sim=self)
        self.wM = Monitor(sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, interval=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data ------------------------


maxNumber = 50
maxTime = 1000.0   # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean, minutes
Nc = 2             # number of counters
seedVal = 99999

# Experiment   ---------------------------

experi = BankModel()
experi.run(aseed=seedVal)

# Result  ---------------------------------

result = experi.wM.count(), experi.wM.mean()
print("Average wait for %3d completions was %5.3f minutes." % result)
```

In previous programs, we have generated the `BankModel` anonymously. Here, we do it differently: we assign the `BankModel` object to the variable `experi` (line 53). This way, we can reference its monitor attribute by `experi`.

wM (line 58). The average waiting time for 50 customers in this 2-counter system is more reliable (i.e., less subject to random simulation effects) than the times we measured before but it is still not sufficiently reliable for real-world decisions. We should also replicate the runs using different random number seeds. The result of this run is:

```
Average wait for  50 completions was 8.941 minutes.
```

Multiple runs

This example demonstrates the power of the object-oriented approach. To get a number of independent measurements we must replicate the runs using different random number seeds. Each replication must be independent of previous ones both in the random numbers and in the data collection so capability of creating independent simulation models within one program is useful.

We do a standard `from SimPy.Simulation import ...` at #1. We define `Source` and `Customer` as sub-classes of `` `Process ``. These differ in detail from the way they are defined in the classic procedure-oriented version of SimPy. Each must refer to the simulation environment it is running in, here the `sim` argument. Thus the current time is returned by the `self.sim.now()` method (at #5

We define a `BankModel` as a sub-class of `Simulation` (at #8) and create an object, `mymodel`, of that class (at #14). A `BankModel` object has a `run` method and this is used for each independent replication (at #16). Note that the `BankModel` is only generated once (line 54). This is sufficient, as the `run` method freshly generates an empty event list, a new counter resource, a new monitor, and a new source. This way, all iterations are independent of each other.

The random number seeds are stored in a list, `seedVals` and the `for` loop walks through this list and executes `mymodel`'s `run` method for each entry to get a set of replications.

```
""" bank12_OO: Multiple runs of the bank with a Monitor"""
from SimPy.Simulation import Simulation, Process, \
    Resource, Monitor, hold, request, release  # 1
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, interval):  # 2
        for i in range(number):
            c = Customer(name="Customer%02d" % (i),
                         sim=self.sim)  # 3
            self.sim.activate(c, c.visit(b=self.sim.k))  # 4
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, b):
        arrive = self.sim.now()  # 5
        yield request, self, b
        wait = self.sim.now() - arrive  # 6
        self.sim.wM.observe(wait)
        tib = expovariate(1.0 / timeInBank)  # 7
        yield hold, self, tib
```

```
        yield release, self, b

# Simulation Model  ---------------------------------


class BankModel(Simulation):  # 8
    def run(self, aseed):
        self.initialize()  # 9
        seed(aseed)
        self.k = Resource(capacity=Nc, name="Clerk",
                          sim=self)  # 10
        self.wM = Monitor(sim=self)  # 11
        s = Source('Source', sim=self)  # 12
        self.activate(s, s.generate(number=maxNumber,
                                    interval=ARRint), at=0.0)
        self.simulate(until=maxTime)  # 13
        return (self.wM.count(), self.wM.mean())


# Experiment data -------------------------
maxNumber = 50
maxTime = 2000.0    # minutes
timeInBank = 12.0   # mean, minutes
ARRint = 10.0       # mean, minutes
Nc = 2              # number of counters
seedVals = [393939, 31555999, 777999555, 319999771]

# Experiment/Result  ----------------------------------

mymodel = BankModel()  # 14
for Sd in seedVals:  # 15
    mymodel.run(aseed=Sd)  # 16
    moni = mymodel.wM  # 17
    print("Avge wait for %3d completions was %6.2f min." %
          (moni.count(), moni.mean()))
```

The results show some variation. Remember, though, that the system is still only operating for 50 customers so the system may not be in steady-state.

```
Avge wait for  50 completions was   3.66 min.
Avge wait for  50 completions was   2.62 min.
Avge wait for  50 completions was   8.97 min.
Avge wait for  50 completions was   5.34 min.
```

3.13.7 Final Remarks

This introduction is too long and the examples are getting longer. There is much more to say about simulation with *SimPy* but no space. I finish with a list of topics for further study:

- **GUI input**. Graphical input of simulation parameters could be an advantage in some cases. *SimPy* allows this and programs using these facilities have been developed (see, for example, program `MM1.py` in the examples in the *SimPy* distribution)
- **Graphical Output**. Similarly, graphical output of results can also be of value, not least in debugging simulation programs and checking for steady-state conditions. SimPlot is useful here.

- **Statistical Output**. The `Monitor` class is useful in presenting results but more powerful methods of analysis are often needed. One solution is to output a trace and read that into a large-scale statistical system such as *R*.
- **Priorities and Reneging in queues**. *SimPy* allows processes to request units of resources under a priority queue discipline (preemptive or not). It also allows processes to renege from a queue.
- **Other forms of Resource Facilities**. *SimPy* has two other resource structures: `Levels` to hold bulk commodities, and `Stores` to contain an inventory of different object types.
- **Advanced synchronization/scheduling commands**. *SimPy* allows process synchronization by events and signals.

3.13.8 Acknowledgements

I thank Klaus Muller, Bob Helmbold, Mukhlis Matti and other developers and users of SimPy for improving this document by sending their comments. I would be grateful for further suggestions or corrections. Please send them to: *vignaux* at *users.sourceforge.net*.

3.13.9 References

- Python website: https://www.python.org
- SimPy website: https://github.com/SimPyClassic/SimPyClassic

3.14 The Bank Tutorial (OO API) Part 2: More examples of SimPy Classic Simulation

Authors G A Vignaux, K G Muller

Date 2010 April

Release 2.3.3

Python-Version 2.7 and later

3.14.1 OO API Bank Tutorial version

This manual is a rework of the *Bank Tutorial Part 2*. Its goal is to show how the simple tutorial models can be written in the advanced OO API.

Note: To contrast the OO API with the procedural SimPy API, the reader should read both "Bank Tutorial Part 2" documents side by side.

3.14.2 Introduction

The first Bank tutorial, The Bank, developed and explained a series of simulation models of a simple bank using SimPy. In various models, customers arrived randomly, queued up to be served at one or several counters, modelled using the Resource class, and, in one case, could choose the shortest among several queues. It demonstrated the use of the Monitor class to record delays and showed how a `model()` mainline for the simulation was convenient to execute replications of simulation runs.

In this extension to The Bank, I provide more examples of SimPy facilities for which there was no room and for some that were developed since it was written. These facilities are generally more complicated than those introduced before. They include queueing with priority, possibly with preemption, reneging, plotting, interrupting, waiting until a condition occurs (`waituntil`) and waiting for events to occur.

Starting with SimPy 2.0 an object-oriented programmer's interface was added to the package and it is this version that is described here. It is quite compatible with the procedural approach. The object-oriented interface, however, can support the process of developing and extending a simulation model better than the procedural approach.

The programs are available without line numbers and ready to go, in directory `bankprograms`. Some have trace statements for demonstration purposes, others produce graphical output to the screen. Let me encourage you to run them and modify them for yourself.

SimPy itself can be obtained from: https://github.com/SimPyClassic/SimPyClassic. It is compatible with Python version 2.7 onwards. The examples in this documentation run with SimPy version 1.5 and later.

This tutorial should be read with the SimPy Manual and CheatsheetOO at your side for reference.

3.14.3 Priority Customers

In many situations there is a system of priority service. Those customers with high priority are served first, those with low priority must wait. In some cases, preemptive priority will even allow a high-priority customer to interrupt the service of one with a lower priority.

SimPy implements priority requests with an extra numerical priority argument in the `yield request` command, higher values meaning higher priority. For this to operate, the requested Resource must have been defined with `qType=PriorityQ`. This require importing the `PriorityQ` class from `SimPy.Simulation`.

Priority Customers without preemption

In the first example, we modify the program with random arrivals, one counter, and a fixed service time (like `bank07.py` in The Bank tutorial) to process a high priority customer. Warning: the `seedVal` value has been changed to `98989` to make the story more exciting.

The modifications are to the definition of the `counter` where we change the `qType` and to the `yield request` command in the `visit` PEM of the customer. We also need to provide each customer with a priority. Since the default is `priority=0` this is easy for most of them.

To observe the priority in action, while all other customers have the default priority of 0, in lines 43 to 44 we create and activate one special customer, `Guido`, with priority 100 who arrives at time `23.0` (line 44). This is to ensure that he arrives after `Customer03`.

The `visit` customer method has a new parameter, `P=0` (line 20) which allows us to set the customer priority.

In lines 39 to 40 the `BankModel` 's resource attribute `k` named `Counter` is defined with `qType=PriorityQ` so that we can request it with priority (line 25) using the statement `yield request,self,self.sim.k,P`

In line 23 we print out the number of customers waiting when each customer arrives.

```
""" bank20_OO: One counter with a priority customer """
from SimPy.Simulation import (Simulation, Process, Resource, PriorityQ, hold,
                              request, release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0, P=0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0, P=0):
        arrive = self.sim.now()       # arrival time
        Nwaiting = len(self.sim.k.waitQ)
        print("%8.3f %s: Queue is %d on arrival" %
              (self.sim.now(), self.name, Nwaiting))
```

```
        yield request, self, self.sim.k, P
        wait = self.sim.now() - arrive  # waiting time
        print("%8.3f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        yield hold, self, timeInBank
        yield release, self, self.sim.k

        print("%8.3f %s: Completed" % (self.sim.now(), self.name))


# Model --------------------------------
class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.k = Resource(name="Counter", unitName="Karen",
                          qType=PriorityQ, sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=5, interval=10.0), at=0.0)
        guido = Customer(name="Guido     ", sim=self)
        self.activate(guido, guido.visit(timeInBank=12.0, P=100), at=23.0)
        self.simulate(until=maxTime)

# Experiment data ------------------------


maxTime = 400.0  # minutes
seedVal = 787878

# Experiment ---------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

The resulting output is as follows. The number of customers in the queue just as each arrives is displayed in the trace. That count does not include any customer in service.

```
   0.000 Customer00: Queue is 0 on arrival
   0.000 Customer00: Waited  0.000
  12.000 Customer00: Completed
  12.083 Customer01: Queue is 0 on arrival
  12.083 Customer01: Waited  0.000
  20.210 Customer02: Queue is 0 on arrival
  23.000 Guido     : Queue is 1 on arrival
  24.083 Customer01: Completed
  24.083 Guido     : Waited  1.083
  34.043 Customer03: Queue is 1 on arrival
  36.083 Guido     : Completed
  36.083 Customer02: Waited 15.873
  48.083 Customer02: Completed
  48.083 Customer03: Waited 14.040
  60.083 Customer03: Completed
  60.661 Customer04: Queue is 0 on arrival
  60.661 Customer04: Waited  0.000
  72.661 Customer04: Completed
```

Reading carefully one can see that when `Guido` arrives `Customer00` has been served and left at `12.000`), `Customer01` is in service and two (customers 02 and 03) are queueing. `Guido` has priority over those waiting

and is served before them at 24.000. When Guido leaves at 36.000, Customer02 starts service.

Priority Customers with preemption

Now we allow Guido to have preemptive priority. He will displace any customer in service when he arrives. That customer will resume when Guido finishes (unless higher priority customers intervene). It requires only a change to one line of the program, adding the argument, preemptable=True to the Resource statement in line 40.

```
""" bank23_OO: One counter with a priority customer with preemption """
from SimPy.Simulation import (Simulation, Process, Resource, PriorityQ, hold,
                              request, release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0, P=0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0, P=0):
        arrive = self.sim.now()       # arrival time
        Nwaiting = len(self.sim.k.waitQ)
        print("%8.3f %s: Queue is %d on arrival" %
              (self.sim.now(), self.name, Nwaiting))

        yield request, self, self.sim.k, P
        wait = self.sim.now() - arrive  # waiting time
        print("%8.3f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
        yield hold, self, timeInBank
        yield release, self, self.sim.k

        print("%8.3f %s: Completed" % (self.sim.now(), self.name))


# Model -----------------------------------
class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.k = Resource(name="Counter", unitName="Karen",
                          qType=PriorityQ, preemptable=True, sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=5, interval=10.0), at=0.0)
        guido = Customer(name="Guido     ", sim=self)
        self.activate(guido, guido.visit(timeInBank=12.0, P=100), at=23.0)
        self.simulate(until=maxTime)
```

```
# Experiment data -------------------------


maxTime = 400.0  # minutes
seedVal = 989898

# Experiment -------- ----------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

Though `Guido` arrives at the same time, `23.000`, he no longer has to wait and immediately goes into service, displacing the incumbent, `Customer01`. That customer had already completed `23.000-12.000 = 11.000` minutes of his service. When `Guido` finishes at `35.000`, `Customer01` resumes service and takes `36.000-35.000 = 1.000` minutes to finish. His total service time is the same as before (`12.000` minutes).

```
  0.000 Customer00: Queue is 0 on arrival
  0.000 Customer00: Waited  0.000
  8.634 Customer01: Queue is 0 on arrival
 12.000 Customer00: Completed
 12.000 Customer01: Waited  3.366
 16.016 Customer02: Queue is 0 on arrival
 19.882 Customer03: Queue is 1 on arrival
 20.246 Customer04: Queue is 2 on arrival
 23.000 Guido     : Queue is 3 on arrival
 23.000 Guido     : Waited  0.000
 35.000 Guido     : Completed
 36.000 Customer01: Completed
 36.000 Customer02: Waited 19.984
 48.000 Customer02: Completed
 48.000 Customer03: Waited 28.118
 60.000 Customer03: Completed
 60.000 Customer04: Waited 39.754
 72.000 Customer04: Completed
```

3.14.4 Balking and Reneging Customers

Balking occurs when a customer refuses to join a queue if it is too long. Reneging (or, better, abandonment) occurs if an impatient customer gives up while still waiting and before being served.

Balking Customers

Another term for a system with balking customers is one where "blocked customers" are "cleared", termed by engineers a BCC system. This is very convenient analytically in queueing theory and formulae developed using this assumption are used extensively for planning communication systems. The easiest case is when no queueing is allowed.

As an example let us investigate a BCC system with a single server but the waiting space is limited. We will estimate the rate of balking when the maximum number in the queue is set to 1. On arrival into the system the customer must first check to see if there is room. We will need the number of customers in the system or waiting. We could keep a count, incrementing when a customer joins the queue or, since we have a Resource, use the length of the Resource's `waitQ`. Choosing the latter we test (on line 23). If there is not enough room, we balk, incrementing a class variable `Customer.numBalking` at line 32 to get the total number balking during the run.

```
""" bank24_OO. BCC system with several counters """
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, meanTBA):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit())
            t = expovariate(1.0 / meanTBA)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives,  is served and leaves """

    def visit(self):
        arrive = self.sim.now()
        print("%8.4f %s: Here I am " % (self.sim.now(), self.name))
        if len(self.sim.k.waitQ) < maxInQueue:      # the test
            yield request, self, self.sim.k
            wait = self.sim.now() - arrive
            print("%8.4f %s: Wait %6.3f" % (self.sim.now(), self.name, wait))
            tib = expovariate(1.0 / timeInBank)
            yield hold, self, tib
            yield release, self, self.sim.k
            print("%8.4f %s: Finished  " % (self.sim.now(), self.name))
        else:
            Customer.numBalking += 1
            print("%8.4f %s: BALKING   " % (self.sim.now(), self.name))
```

```
# Model
class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        Customer.numBalking = 0
        self.k = Resource(capacity=numServers,
                          name="Counter", unitName="Clerk", sim=self)
        s = Source('Source', sim=self)
        self.activate(s, s.generate(number=maxNumber, meanTBA=ARRint), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -----------------------------


timeInBank = 12.0  # mean, minutes
ARRint = 10.0      # mean interarrival time, minutes
numServers = 1     # servers
maxInSystem = 2    # customers
maxInQueue = maxInSystem - numServers

maxNumber = 8
maxTime = 4000.0  # minutes
theseed = 212121

# Experiment ------------------------------------

mymodel = BankModel()
mymodel.run(aseed=theseed)
# Results ---------------------------------------

nb = float(Customer.numBalking)
print("balking rate is %8.4f per minute" % (nb / mymodel.now()))
```

The resulting output for a run of this program showing balking occurring is given below:

```
  0.0000 Customer00: Here I am
  0.0000 Customer00: Wait  0.000
  4.3077 Customer01: Here I am
  5.6957 Customer02: Here I am
  5.6957 Customer02: BALKING
  6.9774 Customer03: Here I am
  6.9774 Customer03: BALKING
  8.2476 Customer00: Finished
  8.2476 Customer01: Wait  3.940
 21.1312 Customer04: Here I am
 22.4840 Customer01: Finished
 22.4840 Customer04: Wait  1.353
 23.0923 Customer05: Here I am
 23.1537 Customer06: Here I am
 23.1537 Customer06: BALKING
 36.0653 Customer04: Finished
 36.0653 Customer05: Wait 12.973
 38.4851 Customer07: Here I am
 53.1056 Customer05: Finished
 53.1056 Customer07: Wait 14.620
 60.3558 Customer07: Finished
balking rate is   0.0497 per minute
```

When `Customer02` arrives, numbers 00 is already in service and 01 is waiting. There is no room so 02 balks. By the vagaries of exponential random numbers, 00 takes a very long time to serve (55.0607 minutes) so the first one to find room is number 07 at 73.0765.

Reneging (or abandoning) Customers

Often in practice an impatient customer will leave the queue before being served. SimPy can model this *reneging* behaviour using a *compound yield statement*. In such a statement there are two yield clauses. An example is:

```
yield (request,self,counter),(hold,self,maxWaitTime)
```

The first tuple of this statement is the usual `yield request`, asking for a unit of `counter` Resource. The process will either get the unit immediately or be queued by the Resource. The second tuple is a reneging clause which has the same syntax as a `yield hold`. The requesting process will renege if the wait exceeds `maxWaitTime`.

There is a complication, though. The requesting PEM must discover what actually happened. Did the process get the resource or did it renege? This involves a *mandatory* test of `self.acquired(`*resource*`)`. In our example, this test is in line 26.

```
""" bank21_OO: One counter with impatient customers """
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly """

    def generate(self, number, interval):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=15.0))
            t = expovariate(1.0 / interval)
            yield hold, self, t


class Customer(Process):
    """ Customer arrives, is served and  leaves """

    def visit(self, timeInBank=0):
        arrive = self.sim.now()        # arrival time
        print("%8.3f %s: Here I am     " % (self.sim.now(), self.name))

        yield (request, self, self.sim.counter), (hold, self, maxWaitTime)
        wait = self.sim.now() - arrive  # waiting time
        if self.acquired(self.sim.counter):
            print("%8.3f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))
            yield hold, self, timeInBank
            yield release, self, self.sim.counter
            print("%8.3f %s: Completed" % (self.sim.now(), self.name))
        else:
            print("%8.3f %s: Waited %6.3f. I am off" %
                  (self.sim.now(), self.name, wait))
```

```
# Model  ---------------------------------
class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.counter = Resource(name="Karen", sim=self)
        source = Source('Source', sim=self)
        self.activate(source,
                      source.generate(number=5, interval=10.0), at=0.0)
        self.simulate(until=maxTime)

# Experiment data ------------------------


maxTime = 400.0     # minutes
maxWaitTime = 12.0  # minutes. maximum time to wait
seedVal = 989898

# Experiment  ---------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

```
  0.000 Customer00: Here I am
  0.000 Customer00: Waited  0.000
  8.634 Customer01: Here I am
 15.000 Customer00: Completed
 15.000 Customer01: Waited  6.366
 16.016 Customer02: Here I am
 19.882 Customer03: Here I am
 20.246 Customer04: Here I am
 28.016 Customer02: Waited 12.000. I am off
 30.000 Customer01: Completed
 30.000 Customer03: Waited 10.118
 32.246 Customer04: Waited 12.000. I am off
 45.000 Customer03: Completed
```

`Customer01` arrives after 00 but has only 12 minutes patience. After that time in the queue (at time 14.166) he abandons the queue to leave 02 to take his place. 03 also abandons. 04 finds an empty system and takes the server without having to wait.

3.14.5 Processes

In some simulations it is valuable for one SimPy Process to interrupt another. This can only be done when the *victim* is "active"; that is when it has an event scheduled for it. It must be executing a `yield hold` statement.

A process waiting for a resource (after a `yield request` statement) is passive and cannot be interrupted by another. Instead the `yield waituntil` and `yield waitevent` facilities have been introduced to allow processes to wait for conditions set by other processes.

Interrupting a Process.

`Klaus` goes into the bank to talk to the manager. For clarity we ignore the counters and other customers. During his conversation his cellphone rings. When he finishes the call he continues the conversation.

In this example, `call` is an object of the `Call` Process class whose only purpose is to make the cellphone ring after a delay, `timeOfCall`, an argument to its `ring` PEM (line 26).

`klaus`, a `Customer`, is interrupted by the call (line 29). He is in the middle of a `yield hold` (line 12). When he exits from that command it is as if he went into a trance when talking to the bank manager. He suddenly wakes up and must check (line 13) to see whether has finished his conversation (if there was no call) or has been interrupted.

If `self.interrupted()` is `False` he was not interrupted and leaves the bank (line 21) normally. If it is `True`, he was interrupted by the call, remembers how much conversation he has left (line 14), resets the interrupt (line 15) and then deals with the call. When he finishes (line 19) he can resume the conversation, with, now we assume, a thoroughly irritated bank manager v(line 20).

```
""" bank22_OO: An interruption by a phone call """
from SimPy.Simulation import Simulation, Process, hold

# Model components ------------------------


class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self, timeInBank, onphone):
        print("%7.4f %s: Here I am" % (self.sim.now(), self.name))
        yield hold, self, timeInBank
        if self.interrupted():
            timeleft = self.interruptLeft
            self.interruptReset()
            print("%7.4f %s: Excuse me" % (self.sim.now(), self.name))
            print("%7.4f %s: Hello! I'll call back" %
                  (self.sim.now(), self.name))
            yield hold, self, onphone
            print("%7.4f %s: Sorry, where were we?" %
                  (self.sim.now(), self.name))
            yield hold, self, timeleft
        print("%7.4f %s: I must leave" % (self.sim.now(), self.name))


class Call(Process):
    """ Cellphone call arrives and interrupts """

    def ring(self, klaus, timeOfCall):
        yield hold, self, timeOfCall
        print("%7.4f Ringgg!" % (self.sim.now()))
        self.interrupt(klaus)
```

```
# Model ----------------------------------


class BankModel(Simulation):
    def run(self):
        """ PEM """
        klaus = Customer(name="Klaus", sim=self)
        self.activate(klaus, klaus.visit(timeInBank, onphone))
        call = Call(sim=self)
        self.activate(call, call.ring(klaus, timeOfCall))
        self.simulate(until=maxTime)

# Experiment data ------------------------


timeInBank = 20.0
timeOfCall = 9.0
onphone = 3.0
maxTime = 100.0

# Experiment  ------------------------------
mymodel = BankModel()
mymodel.run()
```

```
 0.0000 Klaus: Here I am
 9.0000 Ringgg!
 9.0000 Klaus: Excuse me
 9.0000 Klaus: Hello! I'll call back
12.0000 Klaus: Sorry, where were we?
23.0000 Klaus: I must leave
```

As this has no random numbers the results are reasonably clear: the interrupting call occurs at 9.0. It takes `klaus` 3 minutes to listen to the message and he resumes the conversation with the bank manager at 12.0. His total time of conversation is 9.0 + 11.0 = 20.0 minutes as it would have been if the interrupt had not occurred.

`waituntil` the Bank door opens

Customers arrive at random, some of them getting to the bank before the door is opened by a doorman. They wait for the door to be opened and then rush in and queue to be served. The door is modeled by an attribute `door` of `BankModel`.

This model uses the `waituntil` yield command. In the program listing the door is initially closed (line 58) and a method to test if it is open is defined at line 54.

The `Doorman` class is defined starting at line 7 and the single `doorman` is created and activated at at lines 59 and 60. The doorman waits for an average 10 minutes (line 11) and then opens the door.

The `Customer` class is defined at 24 and a new customer prints out `Here I am` on arrival. If the door is still closed, he adds `but the door is shut` and settles down to wait (line 35), using the `yield waituntil` command. When the door is opened by the doorman the `dooropen` state is changed and the customer (and all others waiting for the door) proceed. A customer arriving when the door is open will not be delayed.

```
"""bank14_OO: *waituntil* the Bank door opens"""
from SimPy.Simulation import (Simulation, Process, Resource, hold, waituntil,
                              request, release)
from random import expovariate, seed
```

```
# Model components ------------------------


class Doorman(Process):
    """ Doorman opens the door"""

    def openthedoor(self):
        """ He will open the door when he arrives"""
        yield hold, self, expovariate(1.0 / 10.0)
        self.sim.door = 'Open'
        print("%7.4f Doorman: Ladies and "
              "Gentlemen! You may all enter." % (self.sim.now()))


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank=10):
        arrive = self.sim.now()

        if self.sim.dooropen():
            msg = ' and the door is open.'
        else:
            msg = ' but the door is shut.'
        print("%7.4f %s: Here I am%s" % (self.sim.now(), self.name, msg))

        yield waituntil, self, self.sim.dooropen

        print("%7.4f %s: I can  go in!" % (self.sim.now(), self.name))
        wait = self.sim.now() - arrive
        print("%7.4f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))

        yield request, self, self.sim.counter
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.sim.counter

        print("%7.4f %s: Finished    " % (self.sim.now(), self.name))

# Model  ----------------------------------


class BankModel(Simulation):
    def dooropen(self):
        return self.door == 'Open'

    def run(self, aseed):
```

```
        """ PEM """
        seed(aseed)
        self.counter = Resource(capacity=1, name="Clerk", sim=self)
        self.door = 'Shut'
        doorman = Doorman(sim=self)
        self.activate(doorman, doorman.openthedoor())
        source = Source(sim=self)
        self.activate(source,
                      source.generate(number=5, rate=0.1), at=0.0)
        self.simulate(until=400.0)

# Experiment data ------------------------


maxTime = 2000.0   # minutes
seedVal = 393939

# Experiment  ----------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

An output run for this programs shows how the first three customers have to wait until the door is opened.

```
 0.0000 Customer00: Here I am but the door is shut.
 1.1489 Doorman: Ladies and Gentlemen! You may all enter.
 1.1489 Customer00: I can  go in!
 1.1489 Customer00: Waited  1.149
 6.5691 Customer00: Finished
 8.3438 Customer01: Here I am and the door is open.
 8.3438 Customer01: I can  go in!
 8.3438 Customer01: Waited  0.000
15.5704 Customer02: Here I am and the door is open.
15.5704 Customer02: I can  go in!
15.5704 Customer02: Waited  0.000
21.2664 Customer03: Here I am and the door is open.
21.2664 Customer03: I can  go in!
21.2664 Customer03: Waited  0.000
21.9473 Customer04: Here I am and the door is open.
21.9473 Customer04: I can  go in!
21.9473 Customer04: Waited  0.000
27.6401 Customer01: Finished
56.5248 Customer02: Finished
57.3640 Customer03: Finished
77.3587 Customer04: Finished
```

Wait for the doorman to give a signal: `waitevent`

Customers arrive at random, some of them getting to the bank before the door is open. This is controlled by an automatic machine called the doorman which opens the door only at intervals of 30 minutes (it is a very secure bank). The customers wait for the door to be opened and all those waiting enter and proceed to the counter. The door is closed behind them.

This model uses the `yield waitevent` command which requires a `SimEvent` attribute for `BankModel` to be defined (line 56). The `Doorman` class is defined at line 7 and the `doorman` is created and activated at at labels 56 and 57. The doorman waits for a fixed time (label 12) and then tells the customers that the door is open. This is achieved

on line 13 by signalling the `dooropen` event.

The `Customer` class is defined at 24 and in its PEM, when a customer arrives, he prints out `Here I am`. If the door is still closed, he adds *"but the door is shut"* and settles down to wait for the door to be opened using the `yield waitevent` command (line 34). When the door is opened by the doorman (that is, he sends the `dooropen.signal()` the customer and any others waiting may proceed.

```
""" bank13_OO: Wait for the doorman to give a signal: *waitevent*"""
from SimPy.Simulation import (Simulation, Process, Resource, SimEvent, hold,
                              request, release, waitevent)
from random import *

# Model components ------------------------


class Doorman(Process):
    """ Doorman opens the door"""

    def openthedoor(self):
        """ He will opens the door at fixed intervals"""
        for i in range(5):
            yield hold, self, 30.0
            self.sim.dooropen.signal()
            print("%7.4f You may enter" % (self.sim.now()))


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank=10):
        arrive = self.sim.now()

        if self.sim.dooropen.occurred:
            msg = '.'
        else:
            msg = ' but the door is shut.'
        print("%7.4f %s: Here I am%s" % (self.sim.now(), self.name, msg))
        yield waitevent, self, self.sim.dooropen

        print("%7.4f %s: The door is open!" % (self.sim.now(), self.name))

        wait = self.sim.now() - arrive
        print("%7.4f %s: Waited %6.3f" % (self.sim.now(), self.name, wait))

        yield request, self, self.sim.counter
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.sim.counter
```

```
        print("%7.4f %s: Finished    " % (self.sim.now(), self.name))

# Model  ----------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.dooropen = SimEvent("Door Open", sim=self)
        self.counter = Resource(1, name="Clerk", sim=self)
        doorman = Doorman(sim=self)
        self.activate(doorman, doorman.openthedoor())
        source = Source(sim=self)
        self.activate(source,
                      source.generate(number=5, rate=0.1), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 400.0  # minutes
seedVal = 232323

# Experiment  ----------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)
```

An output run for this programs shows how the first three customers have to wait until the door is opened.

```
 0.0000 Customer00: Here I am but the door is shut.
13.6767 Customer01: Here I am but the door is shut.
13.9068 Customer02: Here I am but the door is shut.
30.0000 You may enter
30.0000 Customer02: The door is open!
30.0000 Customer02: Waited 16.093
30.0000 Customer01: The door is open!
30.0000 Customer01: Waited 16.323
30.0000 Customer00: The door is open!
30.0000 Customer00: Waited 30.000
34.0411 Customer03: Here I am but the door is shut.
40.8095 Customer04: Here I am but the door is shut.
55.4721 Customer02: Finished
57.2363 Customer01: Finished
60.0000 You may enter
60.0000 Customer04: The door is open!
60.0000 Customer04: Waited 19.190
60.0000 Customer03: The door is open!
60.0000 Customer03: Waited 25.959
77.0409 Customer00: Finished
90.0000 You may enter
104.8327 Customer04: Finished
118.4142 Customer03: Finished
120.0000 You may enter
150.0000 You may enter
```

3.14.6 Monitors

Monitors (and Tallys) are used to track and record values in a simulation. They store a list of [time,value] pairs, one pair being added whenever the `observe` method is called. A particularly useful characteristic is that they continue to exist after the simulation has been completed. Thus further analysis of the results can be carried out.

Monitors have a set of simple statistical methods such as `mean` and `var` to calculate the average and variance of the observed values – useful in estimating the mean delay, for example.

They also have the `timeAverage` method that calculates the time-weighted average of the recorded values. It determines the total area under the time~value graph and divides by the total time. This is useful for estimating the average number of customers in the bank, for example. There is an *important caveat* in using this method. To estimate the correct time average you must certainly `observe` the value (say the number of customers in the system) whenever it changes (as well as at any other time you wish) but, and this is important, observing the *new* value. The *old* value was recorded earlier. In practice this means that if we wish to observe a changing value, `n`, using the Monitor, `Mon`, we must keep to the the following pattern:

```
n = n+1
Mon.observe(n,self.sim.now())
```

Thus you make the change (not only increases) and *then* observe the new value. Of course the simulation time `now()` has not changed between the two statements.

Plotting a Histogram of Monitor results

A Monitor can construct a histogram from its data using the `histogram` method. In this model we monitor the time in the system for the customers. This is calculated for each customer in line 29, using the arrival time saved in line 19. We create the Monitor attribute of `BankModel`, `Mon`, at line 39 and the times are `observed` at line 30.

The histogram is constructed from the Monitor, after the simulation has finished, at line 58. The SimPy SimPlot package allows simple plotting of results from simulations. Here we use the SimPlot `plotHistogram` method. The plotting routines appear in lines 60-64. The `plotHistogram` call is in line 61.

```
"""bank17_OO: Plotting a Histogram of Monitor results"""
from SimPy.Simulation import (Simulation, Process, Resource, Monitor, hold,
                              request, release)
from SimPy.SimPlot import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank):
        arrive = self.sim.now()
```

```
        # print("%8.4f %s: Arrived      "%(now(), self.name))

        yield request, self, self.sim.counter
        # print("%8.4f %s: Got counter "%(now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.sim.counter

        # print("%8.4f %s: Finished    " % (now(), self.name))
        t = self.sim.now() - arrive
        self.sim.Mon.observe(t)

# Model  ---------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.counter = Resource(1, name="Clerk", sim=self)
        self.Mon = Monitor('Time in the Bank', sim=self)
        source = Source(sim=self)
        self.activate(source,
                      source.generate(number=20, rate=0.1), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 400.0   # minutes

N = 0
seedVal = 393939

# Experiment  -----------------------------

modl = BankModel()
modl.run(aseed=seedVal)

# Output ---------------------------------
Histo = modl.Mon.histogram(low=0.0, high=200.0, nbins=20)

plt = SimPlot()
plt.plotHistogram(Histo, xlab='Time (min)',
                  title="Time in the Bank",
                  color="red", width=2)
plt.mainloop()
```

Monitoring a Resource

Now consider observing the number of customers waiting or executing in a Resource. Because of the need to `observe` the value after the change but at the same simulation instant, it is impossible to use the length of the Resource's `waitQ` directly with a Monitor defined outside the Resource. Instead Resources can be set up with built-in Monitors.

Here is an example using a Monitored Resource. We intend to observe the average number waiting and active in the `counter` resource. `counter` is defined at line 35 as a `BankModel` attribute and we have set `monitored=True`.

This establishes two Monitors: `waitMon`, to record changes in the numbers waiting and `actMon` to record changes in the numbers active in the `counter`. We need make no further change to the operation of the program as monitoring is then automatic. No `observe` calls are necessary.

After completion of the `run` method, we calculate the `timeAverage` of both `waitMon` and `actMon` (lines 53-54). These can then be printed at the end of the program (line 55).

```
"""bank15_OO: Monitoring a Resource"""
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from random import *

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(
                timeInBank=12.0, counter=self.sim.counter))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self, timeInBank, counter):
        arrive = self.sim.now()
        print("%8.4f %s: Arrived     " % (self.sim.now(), self.name))

        yield request, self, counter
        print("%8.4f %s: Got counter " % (self.sim.now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, counter

        print("%8.4f %s: Finished    " % (self.sim.now(), self.name))

# Model  ----------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.counter = Resource(capacity=1, name="Clerk",
                                monitored=True, sim=self)
        source = Source(sim=self)
        self.activate(source,
                      source.generate(number=5, rate=0.1), at=0.0)
        self.simulate(until=maxTime)

        return

# Experiment data -------------------------
```

```
maxTime = 400.0    # minutes
seedVal = 393939

# Experiment  ---------------------------------

modl = BankModel()
modl.run(aseed=seedVal)

nrwaiting = modl.counter.waitMon.timeAverage()
nractive = modl.counter.actMon.timeAverage()
print('Average waiting = %6.4f\nAverage active  = %6.4f\n' %
      (nrwaiting, nractive))
```

Plotting from Resource Monitors

Like all Monitors, `waitMon` and `actMon` in a monitored Resource contain information that enables us to graph the output. Alternative plotting packages can be used; here we use the simple `SimPy.SimPlot` package just to graph the number of customers waiting for the counter. The program is a simple modification of the one that uses a monitored Resource.

The SimPlot package is imported at line 3. No major changes are made to the main part of the program except that I commented out the print statements. The changes occur in the `run` method from lines 38 to 39. The simulation now generates and processes 20 customers (line 39). The Monitors of the `counter` Resource attribute still exist when the simulation has terminated.

The additional plotting actions take place in lines 54 to 57. Line 55-56 construct a step plot and graphs the number in the waiting queue as a function of time. `waitMon` is primarily a list of *[time,value]* pairs which the `plotStep` method of the SimPlot object, `plt` uses without change. On running the program the graph is plotted; the user has to terminate the plotting `mainloop` on the screen.

```
"""bank16_OO: Plotting from  Resource Monitors"""
from SimPy.Simulation import (Simulation, Process, Resource, hold, request,
                              release)
from SimPy.SimPlot import *
from random import expovariate, seed

# Model components ------------------------


class Source(Process):
    """ Source generates customers randomly"""

    def generate(self, number, rate):
        for i in range(number):
            c = Customer(name="Customer%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(timeInBank=12.0))
            yield hold, self, expovariate(rate)


class Customer(Process):
    """ Customer arrives,  is served and leaves """

    def visit(self, timeInBank):
        arrive = self.sim.now()
        # print("%8.4f %s: Arrived     " % (now(), self.name))
```

```
        yield request, self, self.sim.counter
        # print("%8.4f %s: Got counter " % (now(), self.name))
        tib = expovariate(1.0 / timeInBank)
        yield hold, self, tib
        yield release, self, self.sim.counter

        # print("%8.4f %s: Finished    " % (now(), self.name))

# Model --------------------------------


class BankModel(Simulation):
    def run(self, aseed):
        """ PEM """
        seed(aseed)
        self.counter = Resource(1, name="Clerk", monitored=True, sim=self)
        source = Source(sim=self)
        self.activate(source,
                      source.generate(number=20, rate=0.1), at=0.0)
        self.simulate(until=maxTime)

# Experiment data -------------------------


maxTime = 400.0   # minutes
seedVal = 393939

# Experiment ----------------------------------

mymodel = BankModel()
mymodel.run(aseed=seedVal)

# Output --------------------------------------

plt = SimPlot()
plt.plotStep(mymodel.counter.waitMon,
             color="red", width=2)
plt.mainloop()
```

3.14.7 Acknowledgements

I thank Klaus Muller, Bob Helmbold, Mukhlis Matti and the other developers and users of SimPy for improving this document by sending their comments. I would be grateful for any further corrections or suggestions. Please send them to: *vignaux* at *users.sourceforge.net*.

3.14.8 References

- Python website: https://www.python.org
- SimPy homepage: https://github.com/SimPyClassic/SimPyClassic
- The Bank:

3.15 SimPy Course on the Web

An outstanding tutorial on SimPy by Prof. Norm Matloff (U. of California at Davis) can be found on the Web: A Discrete-Event Simulation Course Based on the SimPy Language.

This course material has been developed by Prof. Matloff in his SimPy courses at U. of California. It is being evolved further, so keep going back to this great teaching material!

CHAPTER 4

Interfacing to External Packages

4.1 Publication-quality plot production with matplotlib

This document deals with producing production-quality plots from SimPy simulation output using the **matplotlib** library. matplotlib is known to work on Linux, Unix, MS Windows and OS X platforms. This library is not part of the SimPy distribution and has to be downloaded and installed separately.

Simulation programs normally produce large quantities of output which needs to be visualized, e.g. by plotting. These plots can help with aggregating data, e.g. for detecting trends over time, frequency distributions or determining the warm-up period of a simulation model experiment.

SimPy's SimPlot plotting package is an easy to use, out-of-the-box capability which can produce a full range of plot graphs on the screen and in PostScript format. After installing SimPy, it can be used without installing any other software. It is tightly integrated with SimPy, e.g. its Monitor data collection class.

The SimPlot library is not intended to produce publication-quality plots. If you want to publish your plots in a report or on the web, consider using an external plotting library which can be called rom Python.

4.1.1 About matplotlib

A very popular plotting library for Python is matplotlib. Its capabilities far exceed those of SimPy's SimPlot. This is how matplotlib is described on its home page:

> *"matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (a la matlab or mathematica), web application servers, and six graphical user interface toolkits."*

The matplotlib screenshots (with Python code) at https://matplotlib.org/gallery/index.html show the great range of quality displays the library can produce with little coding. For the investment in time in downloading, installing and learning matplotlib, the SimPy user is rewarded with a powerful plotting capability.

Downloading matplotlib

Extensive installation instructions are provided at https://matplotlib.org/users/installing.html

matplotlib input data

matplotlib takes separate sequences (lists, tuples, arrays) for x- and y-values. SimPlot, on the other hand, plots Monitor instances, i.e., lists of [x,y] lists.

This difference in data structures is easy to overcome in SimPy by using the Monitor functions `yseries` (returning the list of y-data) and `tseries` (returning the list of time- or x-data).

An example from the Bank Tutorial

As an example of how to use matplotlib with SimPy, a modified version of bank12.py from the Bank Tutorial is used here. It produces a line plot of the counter's queue length and a histogram of the customer wait times:

```
""" Based on bank12.py in Bank Tutorial.
This program uses matplotlib. It produces two plots:
- Queue length over time
- Histogram of queue length
"""
import SimPy.Simulation as Simulation
import pylab as pyl
from random import Random


# Model components
class Source(Simulation.Process):
    """ Source generates customers randomly"""

    def __init__(self, seed=333):
        Simulation.Process.__init__(self)
        self.SEED = seed

    def generate(self, number, interval):
        rv = Random(self.SEED)
        for i in range(number):
            c = Customer(name="Customer%02d" % (i,))
            Simulation.activate(c, c.visit(timeInBank=12.0))
            t = rv.expovariate(1.0 / interval)
            yield Simulation.hold, self, t


class Customer(Simulation.Process):
    """ Customer arrives, is served and leaves """

    def __init__(self, name):
        Simulation.Process.__init__(self)
        self.name = name

    def visit(self, timeInBank=0):
        arrive = Simulation.now()
        yield Simulation.request, self, counter
        wait = Simulation.now() - arrive
        wate.observe(y=wait)
```

```
        tib = counterRV.expovariate(1.0 / timeInBank)
        yield Simulation.hold, self, tib
        yield Simulation.release, self, counter


class Observer(Simulation.Process):
    def __init__(self):
        Simulation.Process.__init__(self)

    def observe(self):
        while True:
            yield Simulation.hold, self, 5
            qu.observe(y=len(counter.waitQ))


# Model
def model(counterseed=3939393):
    global counter, counterRV, waitMonitor
    counter = Simulation.Resource(name="Clerk", capacity=1)
    counterRV = Random(counterseed)
    waitMonitor = Simulation.Monitor()
    Simulation.initialize()
    sourceseed = 1133
    source = Source(seed=sourceseed)
    Simulation.activate(source, source.generate(100, 10.0))
    ob = Observer()
    Simulation.activate(ob, ob.observe())
    Simulation.simulate(until=2000.0)


qu = Simulation.Monitor(name="Queue length")
wate = Simulation.Monitor(name="Wait time")
# Experiment data
sourceSeed = 333
# Experiment
model()
# Output
pyl.figure(figsize=(5.5, 4))
pyl.plot(qu.tseries(), qu.yseries())
pyl.title("Bank12: queue length over time",
          fontsize=12, fontweight="bold")
pyl.xlabel("time", fontsize=9, fontweight="bold")
pyl.ylabel("queue length before counter", fontsize=9, fontweight="bold")
pyl.grid(True)
pyl.show()
pyl.savefig("./bank12.png")
print("Saved graph in current directory as bank12.png")

pyl.clf()
n, bins, patches = pyl.hist(qu.yseries(), 10, normed=True)
pyl.title("Bank12: Frequency of counter queue length",
          fontsize=12, fontweight="bold")
pyl.xlabel("queuelength", fontsize=9, fontweight="bold")
pyl.ylabel("frequency", fontsize=9, fontweight="bold")
pyl.grid(True)
pyl.xlim(0, 30)
pyl.show()
pyl.savefig("./bank12histo.png")
```

```
print("Saved graph in current directory as bank12histo.png")
```

Here is the explanation of this program:

Line number and explanation

07 Imports the matplotlib **pylab** module (this import form is needed to avoid namespace clashes with SimPy).

75 Sets the size of the figures following to a width of 5.5 and a height of 4 inches.

76 Plots the series of queue-length values (qu.yseries()) over their observation times series (qu.tseries()).

77 Sets the figure title, its font size, and its font weight.

79 Sets the x-axis label, its font size, and its font weight.

80 Sets the y-axis label, its font size, and its font weight.

81 Gives the graph a grid.

83 Saves the plot under the given name.

86 Clears the current figure (e.g., resets the axes values from the previous plot).

87 Makes a histogram of the queue-length series (qu.series()) with 10 bins. The *normed* parameter makes the frequency counts relative to 1.

88 Sets the title etc.

90 Sets the x-axis label etc.

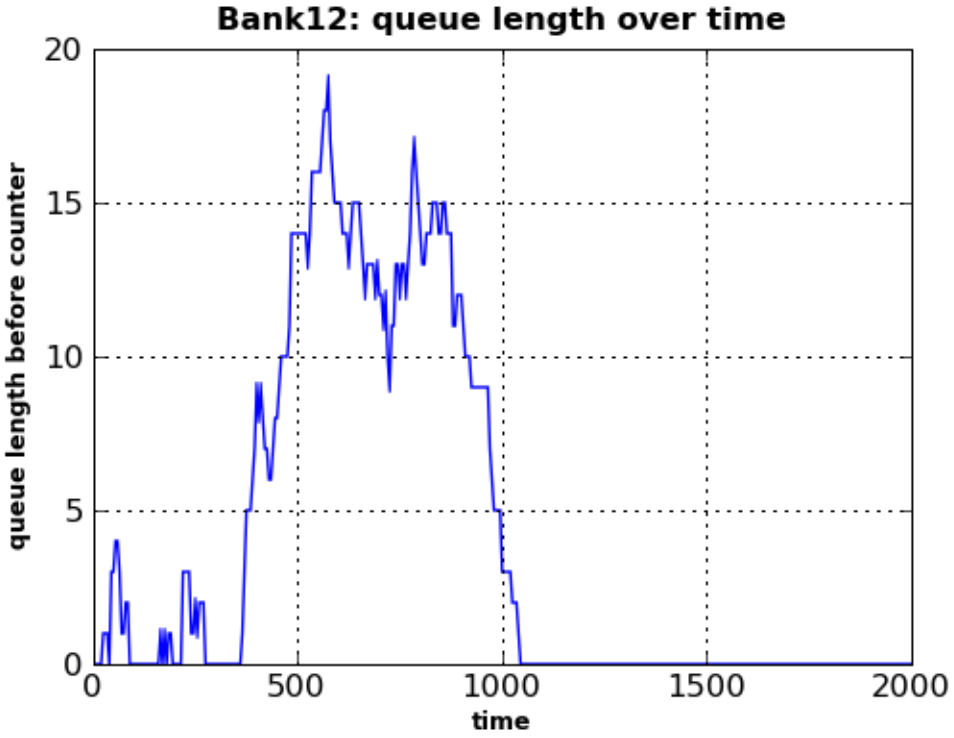
91 Sets the y-axis label etc.

92 Gives the graph a grid.

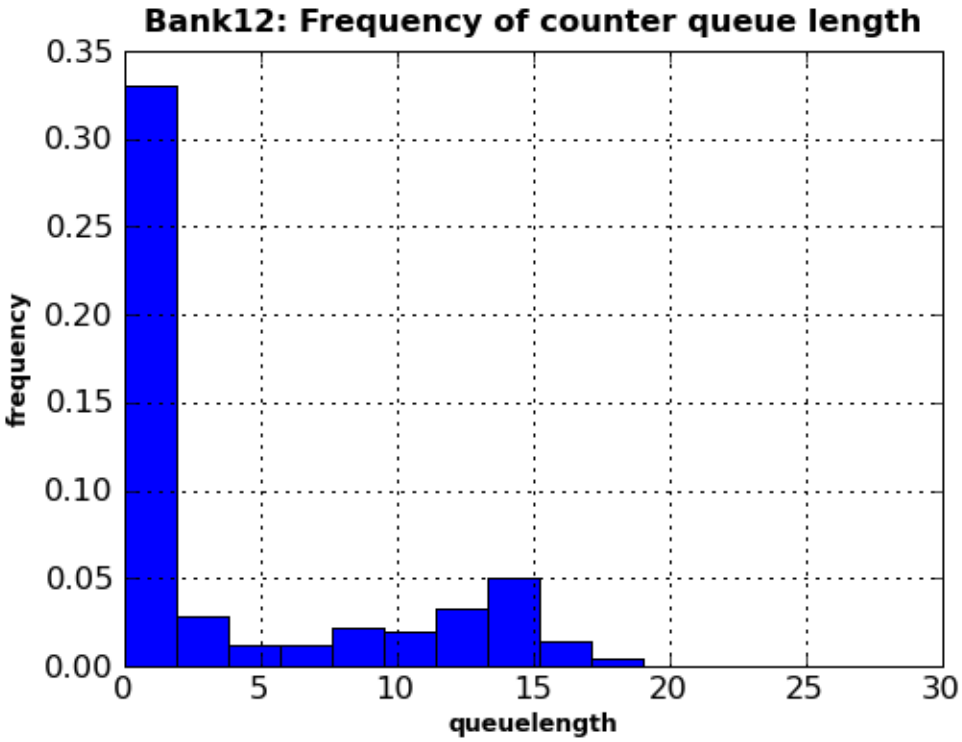
93 Limits the x-axis to the range[0..30].

95 Saves the plot under the given name.

Running the program above results in two PNG files. The first (`bank12.png`) shows the queue length over time:



The second output file (`bank12histo.png`) is a histogram of the customer queue length at the counter:



4.1.2 Conclusion

The small example above already shows the power, flexibility and quality of the graphics capabilities provided by matplotlib. Almost anything (fonts, graph sizes, line types, number of series in one plot, number of subplots in a plot, ...) is under user control by setting parameters or calling functions. Admittedly, it initially takes a lot of reading in the extensive documentation and some experimentation, but the results are definitely worth the effort!

4.2 Running SimPy on Multiple Processors with Parallel Python

Contents

4.2.1 Introduction

With SimPy 2.0, you can easily increase the performance of your simulation by using Parallel Python if you have a larger number of independent processors (multiple CPUs or cores). *Parallel Python* can distribute the execution of your SimPy processes to all cores of your CPU and even to other computers. You should read the PP documentation for further information on how this works.

Please, note that *Parallel Python* is not included in the SimPy distribution and needs to be *downloaded <https://www.parallelpython.com/>* and installed separately.

4.2.2 Examples

Example #1

The files PPExample.txt and PPExampleProcess.txt contain a small example with several car processes. It is important, that processes etc.are not defined in the file that starts the PP job server and executes the jobs, since `ppserver.submit()` only takes functions defined in the same file and module names to import, but no classes.

PPExample.py:

```
"""
Example for SimPy with Parallel Python.
"""

import PPExampleProcess
from SimPy.Simulation import *
import pp


def runSimulation(jobNum, numCars):
    sim = SimPy.Simulation.Simulation()
    cars = []
    for i in range(numCars):
        car = PPExampleProcess.Car(sim, i * jobNum + i)
        sim.activate(car, car.run(), at = 0)
```

```
        cars.append(car)
    sim.simulate(until = 30)


server = pp.Server(ppservers = ())
for i in range(4):
    job = server.submit(
                        runSimulation,
                        (i, 2),
                        (),
                        ('SimPy.Simulation', 'PPExampleProcess'))
    job()
```

PPExampleProcess.py:

```
from SimPy.Simulation import *

class Car(Process):

    def __init__(self, sim, id):
        Process.__init__(self, sim = sim)
        self.id = id

    def run(self):
        while True:
            yield hold, self, 10
            print 'Car #%i at t = %i' % (self.id, self.sim.now())
```

The simulated process in this case is a simple car, that holds for ten steps and then prints the current simulation time. Obviously, each car process is independent from the other ones. Thus if we want to simulate a great number of cars, we can easily distribute the processes to many CPU cores and others computers in our network.

PP pickles everything it sends to other cores/computers. Since SimPy is currently not pickleable, you cannot submit `Simulation.simulate()` to the PPServer. In this example `runSimulation` is defined and submitted to the server. The code within it will be executed on each core/computer. In the example we create four simulation jobs with two cars for each job.

Run PPExample.py to execute the example.

Example #2

Files simulator.txt and processes.txt contain an example that simulates refrigerators in single-thread and in parallel simulation mode.

Class `Simulator` simulates a number of fridges and gets the resulting data.

Class `ParallelSimulator` simulates a number of fridges and gets the resulting data. A number of jobs will be created that use all available CPU cores or even other computers.

To use clustering, ParallelPython needs to be installed on all computers and the server demon "ppserver.py" must be started. The list of the servers' IP addresses must then be passed to the constructor of this class.

Run `simulator.py` to execute this example.

File simulator.py:

```
# coding=utf8
"""
The fridge simulation
```

```
@author: Stefan Scherfke
@contact: stefan.scherfke at uni-oldenburg.de
"""

from time import clock
import logging

from SimPy.Simulation import Simulation, activate, initialize, simulate
import pp

from processes import Fridge, FridgeObserver

log = logging.getLogger('Simulator')

class Simulator(object):
    """
    This class simulates a number of fridges and gets the resulting data.
    """

    def __init__(self, numFridges, tau, aggSteps, duration):
        """
        Setup the simulation with the specified number of fridges.

        Tau specifies the simulation step for each frige. Furthermore the
        observer will collect data each tau. Collected data
        will be aggregated at the end of each aggSteps simulation steps.

        @param numFridges: The number of simulated fridges
        @type numFridges:  unsigned int
        @param tau: simulation step size for collecting data and simulating
                    the fridge
        @type tau:  float
        @param aggSteps: Collected data will be aggregated each aggSteps
                         simulation steps. Signals interval will be
                         tau * aggSteps
        @type aggSteps:  unsigned int
        @param duration: Duration of the simulation in hours
        @type duration:  unsigned int
        """
        log.info('Initializing simulator ...')
        self.simEnd = duration
        self.sim = Simulation()

        fridgeProperties = {'tau': tau}
        self._fridges = []
        for i in range(numFridges):
            fridge = Fridge(self.sim, **fridgeProperties)
            self._fridges.append(fridge)
        self._observer = FridgeObserver(self.sim, self._fridges, tau, aggSteps)

    def simulate(self):
        """
        Initialize the system, start the simulation and return the collected
        data.

        @return: The fridgerators consumption after each aggregation
        """
```

```
        log.info('Running simulation ...')
        self.sim.initialize()
        for fridge in self._fridges:
            self.sim.activate(fridge, fridge.run(), at = 0)
        self.sim.activate(self._observer, self._observer.run(), at = 0)
        self.sim.simulate(until = self.simEnd)

        log.info('Simulation run finished.')
        return self._observer.getData()


class ParallelSimulator(object):
    """
    This class simulates a number of fridges and gets the resulting data.
    Unlike simulator, a number of jobs will be created that use all availale
    CPU cores or even other computers.

    To use clustering, ParallelPython needs to be installed on all computers
    and the server demon "ppserver.py" must be started. The list of the server's
    IPs must then be passed to the constructor of this class.
    """

    def __init__(self, numFridges, tau, aggSteps, duration,
            jobSize = 100, servers = ()):
        """
        Setup the simulation with the specified number of fridges. It will be
        split up in several parallel jobs, each with the specified number of
        jobs.

        Tau specifies the simulation step for each frige. Furthermore the
        observer will collect data each tau. Collected data
        will be aggregated at the end of each aggSteps simulation steps.

        @param numFridges: The number of simulated fridges
        @type numFridges:  unsigned int
        @param tau: simulation step size for collecting data and simulating
        the fridge
        @type tau:  float
        @param aggSteps: Collected data will be aggregated each aggSteps
        simulation steps. Signals interval will be
        tau * aggSteps
        @type aggSteps:  unsigned int
        @param duration: Duration of the simulation
        @type duration:  unsigned int
        @param jobSize: The number of friges per job, defaults to 100.
        @type jobSize: unsigned int
        @param servers: A list of IPs from on which the simulation shall be
                        executed. Defaults to "()" (use only SMP)
        @type servers:  tuple of string
        """
        log.info('Initializing prallel simulation ...')

        self._jobSize = jobSize
        self._servers = servers
        self._numFridges = numFridges
        self._tau = tau
        self._aggSteps = aggSteps
        self.simEnd = duration
```

```
    def simulate(self):
        """
        Create some simulation jobs, run them and retrieve their results.

        @return: The fridgerators consumption after each aggregation
        """
        log.info('Running parallel simulation ...')
        oldLevel = log.getEffectiveLevel() # pp changes the log level :(
        jobServer = pp.Server(ppservers = self._servers)

        # Start the jobs
        remainingFridges = self._numFridges
        jobs = []
        while remainingFridges > 0:
            jobs.append(jobServer.submit(self.runSimulation,
                    (min(self._jobSize, remainingFridges),),
                    (),
                    ("logging", "SimPy.Simulation", "processes")))
            remainingFridges -= self._jobSize
        log.info('Number of jobs for simulation: %d' % len(jobs))

        # Add each job's data
        pSum = [0] * int((60 / self._aggSteps) * self.simEnd)
        for job in jobs:
            data = job()
            for i in range(len(data)):
                pSum[i] += data[i]
        for s in pSum:
            s /= len(jobs)

        log.setLevel(oldLevel)
        log.info('Parallel simulation finished.')
        return pSum

    def runSimulation(self, numFridges):
        """
        Create a job with the specified number of fridges and controllers and
        one observer. Simulate this and return the results.

        @param numFridges: The number of fridges to use for this job
        @type numFridges:  unsigned int
        @return: A list with the aggregated fridge consumption
        """
        sim = SimPy.Simulation.Simulation()
        sim.initialize()

        fridgeProperties = {'tau': self._tau}
        fridges = []
        for i in range(numFridges):
            fridge = processes.Fridge(sim, **fridgeProperties)
            fridges.append(fridge)
            sim.activate(fridge, fridge.run(), at = 0)
        observer = processes.FridgeObserver(sim,
                fridges, self._tau, self._aggSteps)
        sim.activate(observer, observer.run(), at = 0)

        sim.simulate(until = self.simEnd)
```

```
        return observer.getData()


if __name__ == '__main__':
    logging.basicConfig(
            level = logging.INFO,
            format = '%(asctime)s %(levelname)8s: %(name)s: %(message)s')

    numFridges = 5000
    tau = 1./60
    aggStep = 15
    duration = 4 + tau

    sim = Simulator(numFridges, tau, aggStep, duration)
    data = sim.simulate()
    log.info('Results: ' + str(data))

    servers = ()
    sim = ParallelSimulator(numFridges, tau, aggStep, duration, 100, servers)
    data = sim.simulate()
    log.info('Results: ' + str(data))
```

File process.py:

```
# coding=utf8
"""
This file contains classes for simulating, controlling and observing a fridge.

@author: Stefan Scherfke
@contact: stefan.scherfke at uni-oldenburg.de
"""

from math import exp
import logging
import random

from SimPy.Simulation import Process, Simulation, \
        activate, hold, initialize, now, simulate

log = logging.getLogger('Processes')

class Fridge(Process):
    """
    This class represents a simulated fridge.

    It's temperature T for and equidistant series of time steps is computed by
    $T_{i+1} = \epsilon \cdot T_i + (1 - \epsilon) \cdot \left(T^O - \eta
    \cdot \frac{q_i}{A}\right)$ with $\epsilon = e^{-\frac{\tau A}{m_c}}$.
    """

    def __init__(self, sim, T_O = 20.0, A = 3.21, m_c = 15.97, tau = 1.0/60,
                 eta = 3.0, q_i = 0.0, q_max = 70.0,
                 T_i = 5.0, T_range = [5.0, 8.0], noise = False):
        """
        Init all required variables.

        @param sim:       The SimPy simulation this process belongs to
        @type sim:        SimPy.Simulation
```

```
        @param T_O:       Outside temperature
        @param A:         Insulation
        @param m_c:       Thermal mass/thermal storage capacity
        @param tau:       Time span between t_i and t_{i+1}
        @param eta:       Efficiency of the cooling device
        @param q_i:       Initial/current electrical power
        @param q_max:     Power required during cool-down
        @param T_i:       Initial/current temperature
        @param T_range:   Allowed range for T_i
        @param noise:     Add noise to the fridge's parameters, if True
        @type noise:      bool
        """
        Process.__init__(self, sim = sim)
        self.T_O = T_O
        self.A = A
        self.m_c = random.normalvariate(20, 4.5) if noise else m_c
        self.tau = tau
        self.eta = eta
        self.q_i = q_i
        self.q_max = q_max
        self.T_i = random.uniform(T_range[0], T_range[1]) if noise else T_i
        self.T_range = T_range

    def run(self):
        """
        Calculate the fridge's temperature for the current time step.
        """
        while True:
            epsilon = exp(-(self.tau * self.A) / self.m_c)
            self.T_i = epsilon * self.T_i + (1 - epsilon) \
                    * (self.T_O - self.eta * (self.q_i / self.A))
            if self.T_i >= self.T_range[1]:
                self.q_i = self.q_max          # Cool down
            elif self.T_i <= self.T_range[0]:
                self.q_i = 0.0                 # Stop cooling
            log.debug('T_i: %2.2f°C at %.2f' % (self.T_i, self.sim.now()))
            yield hold, self, self.tau

    def coolDown(self):
        """
        Start cooling down now!
        """
        self.q_i = self.q_max


class FridgeObserver(Process):
    """
    This process observes the temperature and power consumption of a set of
    fridges.
    """

    def __init__(self, sim, fridges, tau, aggSteps):
        """
        Init the observer.

        @param sim: The SimPy simulation this process belongs to
        @type sim:  SimPy.Simulation
        @param fridges: A list of fridges to be observed
```

```
        @type fridges: tuple of Fridge
        @param tau: Time interval for observations
        @type tau: float
        @param aggSteps: Specifies after how many timesteps tau the collected
                         data is aggregated and stored.
        @type aggSteps: int
        """
        Process.__init__(self, sim = sim)
        self._fridges = fridges
        self._tau = tau
        self._aggSteps = aggSteps
        self._data = []

    def run(self):
        """
        Start observation
        """
        aggSteps = 0
        consumption = 0
        lastProgUpdate = 0
        while True:
            prog = self.sim.now() * 100 / self.sim._endtime
            if int(prog) > lastProgUpdate:
                log.info('Progress: %d%%' % prog)
                lastProgUpdate = prog
            if (aggSteps >= self._aggSteps):
                log.debug('Aggregating at %.2f' % self.sim.now())
                self._data.append(consumption/self._aggSteps)
                consumption = 0
                aggSteps = 0

            for fridge in self._fridges:
                consumption += fridge.q_i
            aggSteps += 1
            yield hold, self, self._tau

    def getData(self):
        """
        Return the collected data

        @return: a list with the collected data
        """
        return self._data


if __name__ == '__main__':
    logging.basicConfig(
            level = logging.DEBUG,
            format = '%(levelname)-8s %(asctime)s %(name)s: %(message)s')

    tau = 1./60 # Step size 1min
    aggSteps = 15 # Aggregate consumption in 15min blocks
    params = {'tau': tau}

    sim = Simulation()

    fridge = Fridge(sim, **params)
    observer = FridgeObserver(sim, [fridge], tau, aggSteps)
```

```
    sim.activate(fridge, fridge.run(), at = 0)
    sim.activate(observer, observer.run(), at = 0)
    sim.simulate(until = 4 + tau)
    print observer.getData()
```

CHAPTER 5

SimPy Classic Tools

This section contains descriptions of tools written for SimPy Classic which help with its use.

5.1 SimPlot Manual

Authors Klaus Muller <Muller@users.sourceforge.net>

SimPy Classic Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7 and later

Date December 2011

Updated January 2018

Contents

The **SimPlot** plotting library has been developed for SimPy users so that they can produce, view and print simple plots, without having to download and install any other software package.

This manual is aimed at the SimPy applications programmer. It describes the capabilities of SimPlot and its programming interface.

There are several more elaborate Open Source plotting packages downloadable from the Internet which can be used from Python and therefore from SimPy. SimPlot is the "quick and dirty", out-of-the-box plotting package for SimPy. If you need more complex plots or publication-quality graphics, consider using e.g. Matplotlib (<https://matplotlib.org/>_).

5.1.1 Introduction

SimPlot is a basic plotting package based on Tk/Tkinter and designed for use with SimPy. It has been developed from an Open Source plotting package published in John E. Grayson's excellent book 'Python and Tkinter Programming' (ISBN 1-884777-81-3) which in turn was derived from Konrad Hinsen's graph widget for NumPy (Numerical Python).

SimPlot provides for the generation, viewing and Postscript output of a variety of plot types by SimPy programs. The data series of SimPy Monitor instances can be plotted automatically.

SimPlot requires Tk/Tkinter to be installed (for all major operating systems on which Python runs, the Python installation already includes the installation of Tk/Tkinter, so no additional download or installation is required). Test whether Tk/Tkinter is installed by running 'import Tkinter' on the Python interpreter command line. If you don't get an error message, it is.

To write SimPlot-based programs, only a very rudimentary understanding of Tk/Tkinter is required. *This manual does not attempt to teach Tk/Tkinter!*

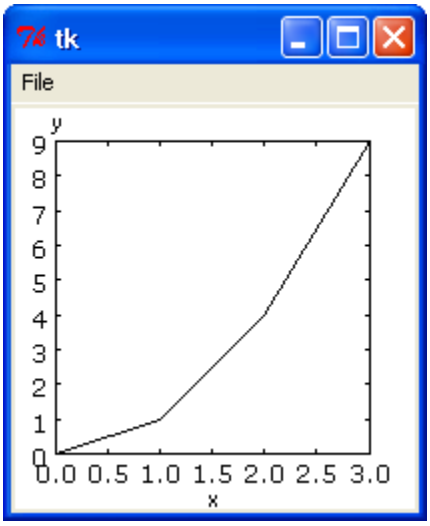
5.1.2 Plotting with SimPlot - an overview

Plot types and capabilities

A simple plot program using SimPlot basically looks like:

```
# Prog1.py
from SimPy.SimPlot import *
plt = SimPlot()
plt.plotLine([[0, 0], [1, 1], [2, 4], [3, 9]])
plt.mainloop()
```

When running this program, the resulting output on Windows is (the outside frame will look different on other platforms):



The program shows the basic structure of any program using SimPlot:

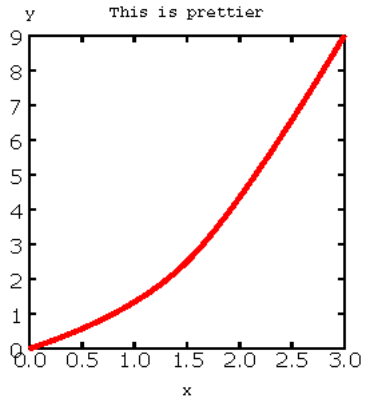
- Line 2 imports the plotting module,
- Line 3 creates and instance of the plotting class,
- Line 4 plots a line in an x/y coordinate system,
- Line 5 starts the main loop of Tk.

The frame also shows a 'File' menu item (when clicked, it offers a submenu item 'Postscript' which allows saving the plot to a Postscript file.

Method *plotline* has many name parameters with default values. Here is an example showing some of them (they will all be discussed further down in this manual:

```
# Prog2.py
from SimPy.SimPlot import *
plt = SimPlot()
plt.plotLine([[0, 0], [1, 1], [2, 4], [3, 9]], title="This is prettier",
             color="red", width=2, smooth=True)
plt.mainloop()
```

This produces the following plot (the outside frame is not shown):



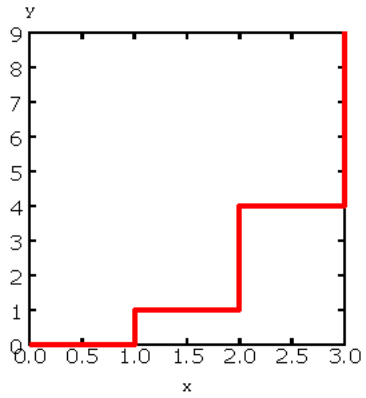
The plot now has a title and the curve is red, wider and smooth.

In addition to *line plots*, there are three other plot-types available in SimPlot, namely *stepped line plots*, *bar charts*, and *scatter diagrams*.

Here are examples of each. First, the stepped line plot:

```
# Prog3.py
from SimPy.SimPlot import *
plt = SimPlot()
plt.plotStep([[0, 0], [1, 1], [2, 4], [3, 9]],
             color="red", width=2)
plt.mainloop()
```

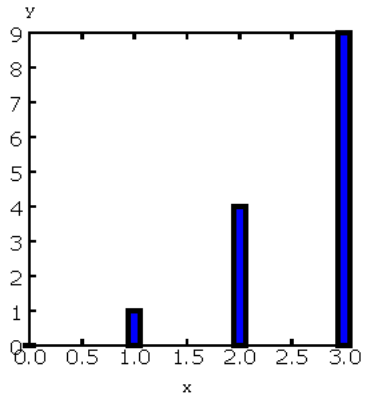
which produces:



A bar chart program:

```
# Prog4.py
from SimPy.SimPlot import *
plt = SimPlot()
plt.plotBars([[0, 0], [1, 1], [2, 4], [3, 9]],
             color="blue", width=2)
plt.mainloop()
```

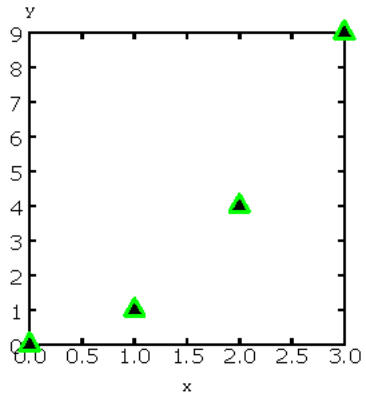
which results in:



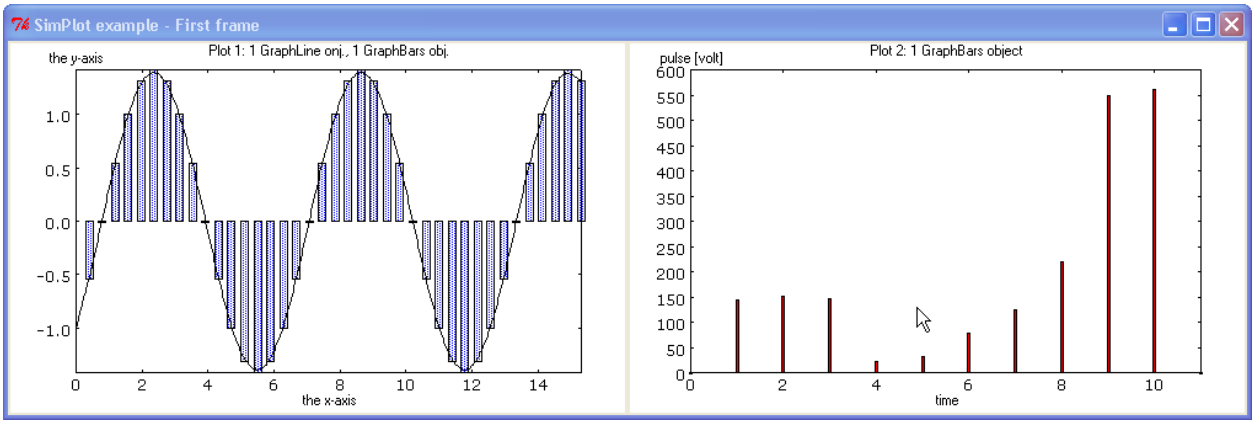
And finally, a scatter diagram:

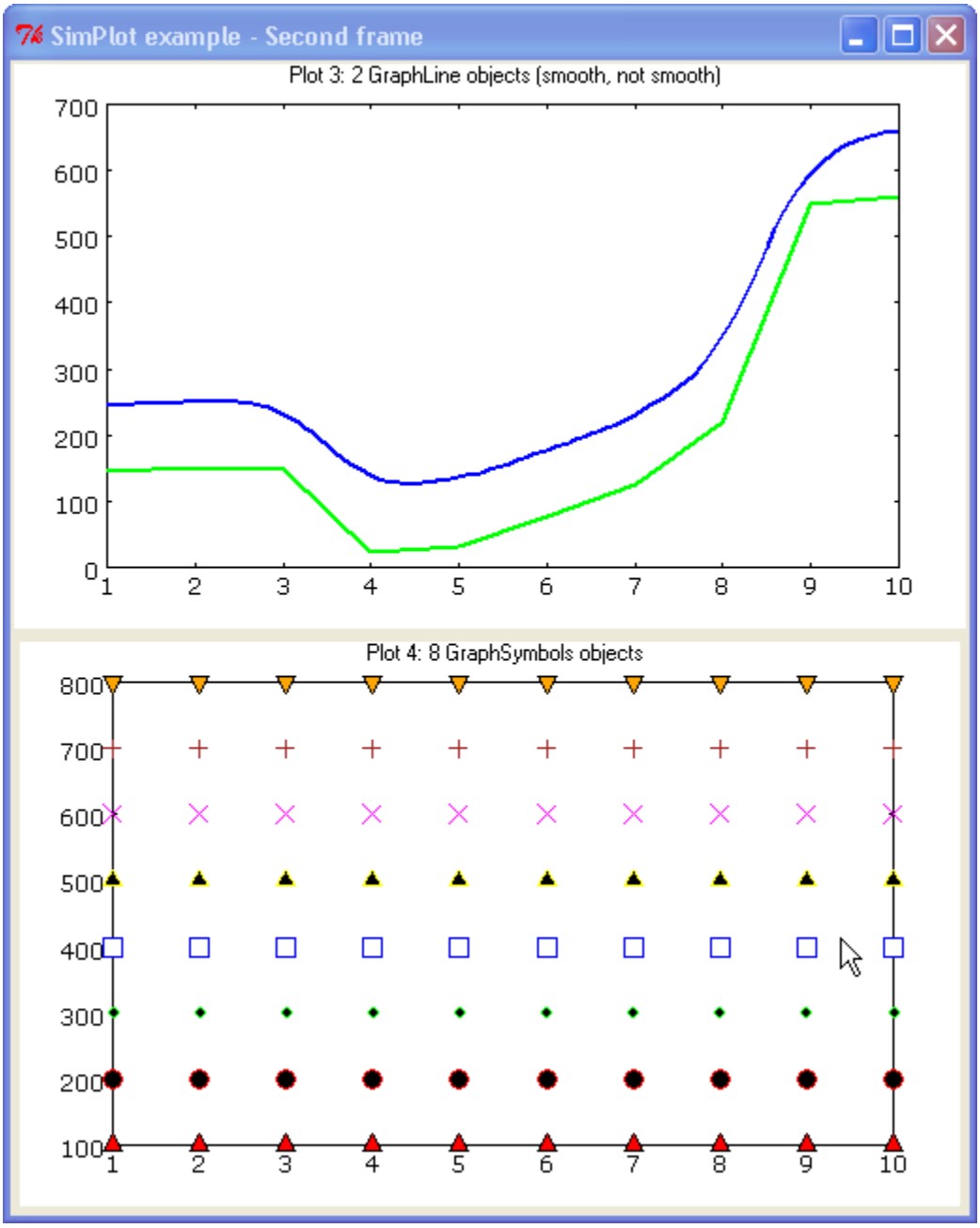
```
# Prog5.py
from SimPy.SimPlot import *
plt = SimPlot()
plt.plotScatter([[0, 0], [1, 1], [2, 4], [3, 9]],
                color="green", size=2, marker='triangle')
plt.mainloop()
```

and its output:



With a bit more involved programming, it is also possible to have several plots in one diagram and to have several diagrams in one Frame (just execute SimPlot.py to get these plots):





Note: In future versions of SimPlot, this part of the API will also be simplified so that it will require significantly less coding.

Plotting Monitor instances

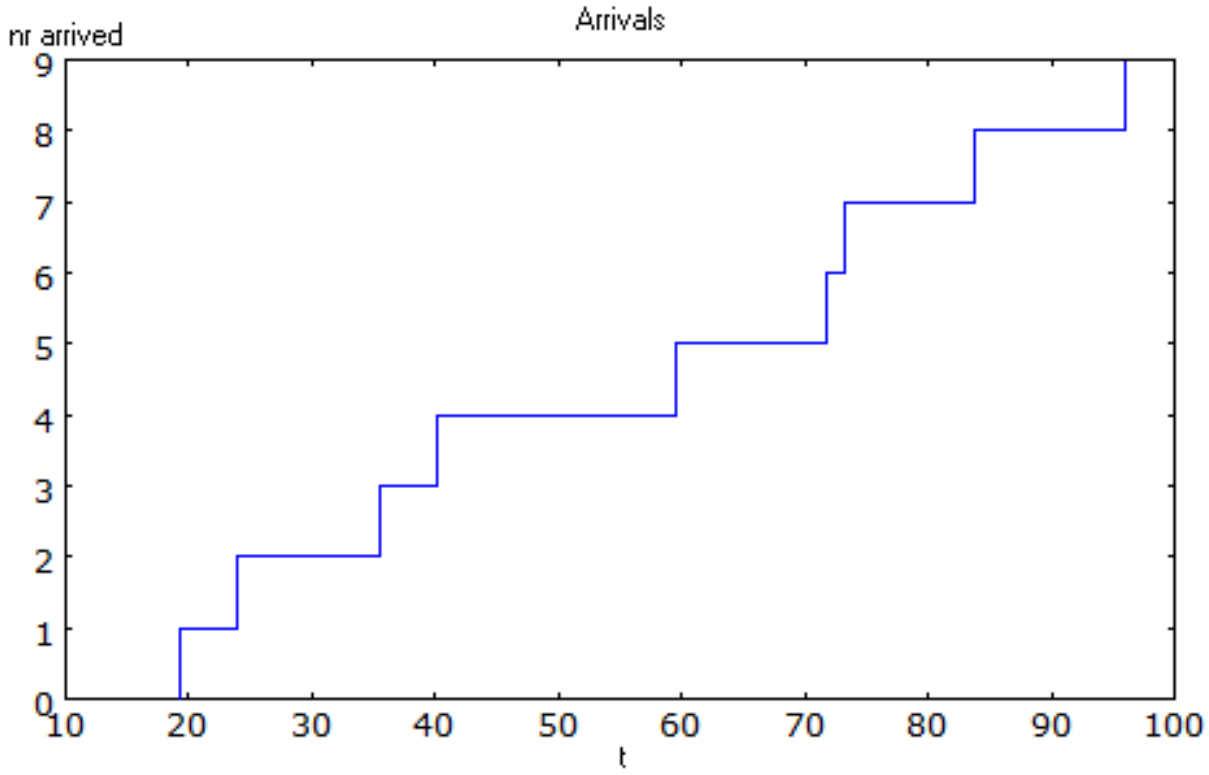
Class *Monitor* is the prime data collection tool for SimPy simulations. SimPlot therefore caters for easy plotting from Monitor intances.

Here is an example of a simple simulation using a Monitor:

```
# Monitorplot.py
from random import uniform
```

```
from SimPy.Simulation import *
from SimPy.Recording import *
from SimPy.SimPlot import *


class Source(Process):
    def __init__(self, monitor):
        Process.__init__(self)
        self.moni = monitor
        self.arrived = 0

    def arrivalGenerator(self):
        while True:
            yield hold, self, uniform(0, 20)
            self.arrived += 1
            self.moni.observe(self.arrived)


initialize()
moni = Monitor(name="Arrivals", ylab="nr arrived")
s = Source(moni)
activate(s, s.arrivalGenerator())
simulate(until=100)

plt = SimPlot()
plt.plotStep(moni, color='blue')
plt.mainloop()
```

This produces:



5.1.3 Simple plotting API

Overview

A SimPlot application program has the following structure:

- instantiation of the **SimPlot** class
- call of one or more plotting methods
- call to the SimPlot instance's **mainloop** method

SimPlot exposes plotting methods at two levels, a *simple API* with limited capabilities but ease of use, and an *advanced API* with SimPlot's full capabilities, but with more involved, verbose programming.

This section deals with the simple API.

SimPlot

class SimPlot

This class provides the plotting capabilities.

plotLine

classmethod SimPlot.**plotLine**(*values*[, *optional parameters*])

Generates a line plot from a list of tuples (or lists) or from a Monitor (any instance which has the attributes 'name', 'tlab', 'ylab').

Parameters

- **values** (*list or Monitor*) – (**mandatory**) a list of two-element lists (or tuples), or a Monitor instance (any instance which has the attributes 'name', 'tlab', 'ylab')
- **windowsize** – the plotting window size in pixels (tuple); default: (500,300)
- **title** (*string*) – the plot title ; if *values* is a Monitor, Monitor name is used if no *title* given
- **width** (*integer or floating point*) – line drawing width, default: 1
- **color** (*Tkinter colour type*) – line colour; default: 'black'
- **smooth** (*boolean*) – if True, makes line smooth; default: "True"
- **background** (*Tkinter colour type*) – colour of plot background; default: 'white'
- **xlab** (*string*) – ** : label on x-axis of plot; if *values* is a Monitor, Monitor *tlab* is taken; default: 'x'
- **ylab** – label on y-axis of plot (string); if *values* is a Monitor, Monitor *ylab* is taken; default: 'y'
- **xaxis** – layout of x-axis (None = omit x-axis; 'automatic' = make x-axis at least long enough to include all x-values, using round values; 'minimal' = have x-axis go *exactly* from minimal to maximal x-value provided; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh); default: 'automatic'
- **yaxis** – layout of y-axis (None = omit y-axis; 'automatic' = make y-axis at least long enough to include all y-values, using round values; 'minimal' = have y-axis go *exactly* from

minimal to maximal y-value provided; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh); default: 'automatic'

Return type Reference to *GraphBase* object which contains the plot.

plotStep

classmethod SimPlot.**plotStep**(*values*[, *optional parameters*])

Generates a step plot from a list of tuples (or lists) or from a Monitor (any instance which has the attributes 'name', 'tlab', 'ylab'). A horizontal line is drawn at a y-value until y changes, creating a step effect.

<variable> = <SimPlotInstance>.plotStep(values[,optional parameters])

param values (**mandatory**) a list of two-element lists (or tuples), or a Monitor instance (any instance which has the attributes 'name', 'tlab', 'ylab')

Optional parameters with defaults:

- **windowsize=(500,300),** : the plotting window size in pixels (tuple)
- **title="** : the plot title (string); if **values** is a Monitor, Monitor name is used if no **title** given
- **width=1** : line drawing width (integer or floating point)
- **color='black'** : line colour (Tkinter colour type; see section on Colours in SimPlot)
- **background='white'** : colour of plot background (Tkinter colour type; see section on Colours in SimPlot)
- **xlab='x'** : label on x-axis of plot (string); if **values** is a Monitor, Monitor *tlab* is taken
- **ylab='y'** : label on y-axis of plot (string); if **values** is a Monitor, Monitor *ylab* is taken
- **xaxis='automatic'** : layout of x-axis (None = omit x-axis; 'automatic' = make x-axis at least long enough to include all x-values, using round values; 'minimal' = have x-axis go *exactly* from minimal to maximal x-value provided; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh)
- **yaxis='automatic'** : layout of y-axis (None = omit y-axis; 'automatic' = make y-axis at least long enough to include all y-values, using round values; 'minimal' = have y-axis go *exactly* from minimal to maximal y-value provided; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh)

Return value: Reference to *GraphBase* object which contains the plot.

plotBars

Generates a bar chart plot from a list of tuples (or lists) or from a Monitor.

Call: <SimPlotInstance>.plotBars(values[,optional parameters])

<variable> = <SimPlotInstance>.plotBars(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples), or a Monitor instance

Optional parameters with defaults:

- **windowsize=(500,300),** : the plotting window size in pixels (tuple)
- **title='** : the plot title (string); if **values** is a Monitor, Monitor name is used if no **title** given
- **width=1** : outline drawing width (integer or floating point)
- **color='black'** : outline colour (Tkinter colour type; see section on Colours in SimPlot)

- **fillcolor='black'** : colour with which bars are filled (Tkinter colour type; see section on Colours in SimPlot)
- **fillstyle='‘** : density of fill (default=100%; Tkinter bitmap)
- **outline='black''** : colour of bar outline ((Tkinter colour type; see section on Colours in SimPlot)
- **background='white'** : colour of plot background (Tkinter colour type; see section on Colours in SimPlot)
- **xlab='x'** : label on x-axis of plot (string); if **values** is a Monitor, Monitor *tlab* is taken
- **ylab='y'** : label on y-axis of plot (string); if **values** is a Monitor, Monitor *ylab* is taken
- **xaxis='automatic'** : layout of x-axis (None = omit x-axis; 'automatic' = make x-axis at least long enough to include all x-values, using round values; 'minimal' = have x-axis go *exactly* from minimal to maximal x-value provided; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh)
- **yaxis='automatic'** : layout of y-axis (None = omit y-axis; 'automatic' = make y-axis at least long enough to include all y-values, using round values; 'minimal' = have y-axis go *exactly* from minimal to maximal y-value provided; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh)

Return value: Reference to *GraphBase* object which contains the plot.

plotHistogram

Generates a histogram plot from a Histogram or a Histogram-like list or tuple. A SimPy Histogram instance is a list with items of two elements. It has n+2 bins of equal width, sorted by the first element, containing integer values == the counts of the bins. The first bin is the 'under' bin, the last the 'over' bin. Histogram objects are produced from Monitor objects by calling the Monitor method *histogram()*.

Call: <SimPlotInstance>.plotHistogram(values[,optional parameters])

<variable> = <SimPlotInstance>.plotHistogram(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples), or a Monitor instance

Optional parameters with defaults:

- **windowsize=(500,300)** : the plotting window size in pixels (tuple)
- **title='‘** : the plot title (string); if **values** is a Monitor, Monitor name is used if no **title** given
- **width=1** : line drawing width (integer or floating point)
- **color='black'** : line colour (Tkinter colour type; see section on Colours in SimPlot)
- **background='white'** : colour of plot background (Tkinter colour type; see section on Colours in SimPlot)
- **xlab='x'** : label on x-axis of plot (string)
- **ylab='y'** : label on y-axis of plot (string)
- **xaxis='automatic'** : layout of x-axis (None = omit x-axis; 'automatic' = make x-axis at least long enough to include all x-values, using round values; 'minimal' = have x-axis go *exactly* from minimal to maximal x-value provided; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh)
- **yaxis='automatic'** : layout of y-axis (None = omit y-axis; 'automatic' = make y-axis at least long enough to include all y-values, using round values; 'minimal' = have y-axis go *exactly* from minimal to maximal y-value provided; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh)

plotScatter

Generates a scatter diagram plot from a list of tuples (or lists) or from a Monitor.

Call: <SimPlotInstance>.plotScatter(values[,optional parameters])

variable = <SimPlotInstance>. plotScatter(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples), or a Monitor instance

Optional parameters with defaults:

- **windowsize=(500,300),** : the plotting window size in pixels (tuple)
- **title='‘** : the plot title (string); if **values** is a Monitor, Monitor name is used if no **title** given
- **marker='circle'** : symbol type (literal; values supported: 'circle, 'dot', 'square', 'triangle, 'triangle_down', 'cross', 'plus')
- **width=1** : line drawing width (integer or floating point)
- **color='black'** : line colour (Tkinter colour type; see section on Colours in SimPlot)
- **fillcolor='black'** : colour with which bars are filled (Tkinter colour type; see section on Colours in SimPlot)
- **fillstyle='‘** : density of fill (default=100%; Tkinter bitmap)
- **outline='black'** : colour of marker outline ((Tkinter colour type; see section on Colours in SimPlot)
- **background='white'** : colour of plot background (Tkinter colour type; see section on Colours in SimPlot)
- **xlab='x'** : label on x-axis of plot (string); if **values** is a Monitor, Monitor *tlab* is taken
- **ylab='y'** : label on y-axis of plot (string); if **values** is a Monitor, Monitor *ylab* is taken
- **xaxis='automatic'** : layout of x-axis (None = omit x-axis; 'automatic' = make x-axis at least long enough to include all x-values, using round values; 'minimal' = have x-axis go *exactly* from minimal to maximal x-value provided; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh)
- **yaxis='automatic'** : layout of y-axis (None = omit y-axis; 'automatic' = make y-axis at least long enough to include all y-values, using round values; 'minimal' = have y-axis go *exactly* from minimal to maximal y-value provided; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh)

Return value: Reference to *GraphBase* object which contains the plot.

postscr

Saves Postscript output from a plot to a file. After e.g. `aPlot=plotLine([0,1],[3,4]), aPlot.postscr("c:\\myplot.ps")` outputs the line plot in Postscript to file c:\myplot.ps.

Call: <plotinstance>.postscr([optional parameter]) (with <plotinstance> being a reference to the *GraphBase* object which contains the plot)

Mandatory parameters: None.

Optional parameters with defaults:

- **"<filename>"** : name of file (complete path) to which Postscript output is written. If omitted, a dialog asking the user for a filename pops up.

Return value: None.

5.1.4 Advanced plotting API

Overview

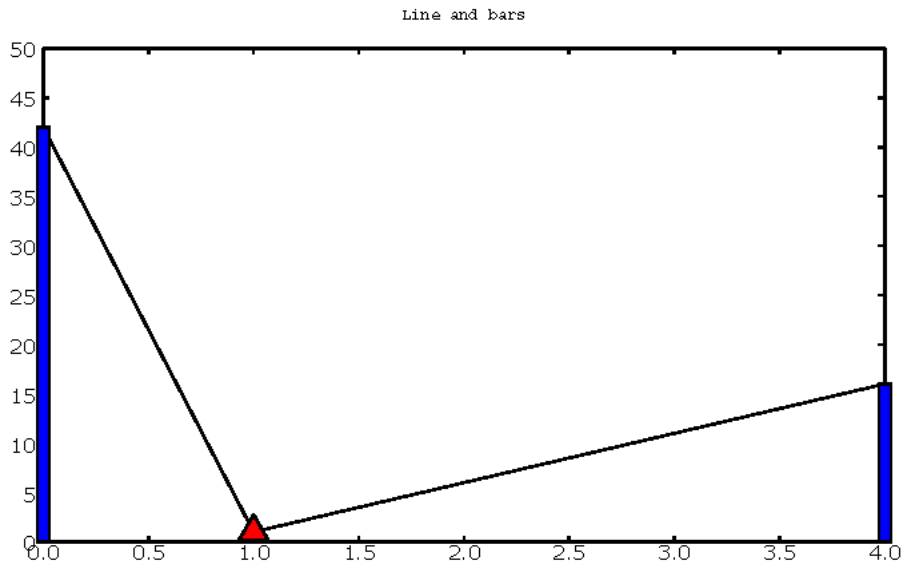
The advanced SimPlot API is more verbose than the simple one, but it offers more flexibility and power. The detailed structure of a program using that API is:

1. make an instance of SimPlot (this initializes Tk and generates a Tk Toplevel container <instance>.root which points at the Tk object.)
2. (optional) make other Tk container(s)
3. (optional) give the container a title
4. make one or more plot objects (the lines or other figures to plot)
5. put the plot objects into a GraphObject (this does the necessary scaling)
6. make a Tk container (e.g. a Frame widget) in the previous container (from step 1 or 2)
7. make a background (with title, axes, frame, etc.) in that container for the GraphObject to be drawn against (i.e., create the graph widget and associate the GraphObject with it)
8. instruct the Tk geometry manager (pack or grid) where to put the background in the Tk container
9. draw the GraphObject against the background
10. instruct the Tk geometry manager concerning the container from step 6
11. (optional) save plot as Postscript file
12. start the Tk mainloop

An example:

```
# AdvancedAPI.py
from SimPy. SimPlot import *
plt = SimPlot()                                    # step 1
plt.root.title("Advanced API example")             # step 3
line = plt.makeLine([[0, 42], [1, 1], [4, 16]])      # step 4
bar = plt.makeBars([[0, 42], [1, 1], [4, 16]],
                   color='blue')                   # step 4
sym = plt.makeSymbols([[1, 1]], marker="triangle",
                      size=3, fillcolor="red")     # step 4
obj = plt.makeGraphObjects([line, bar, sym])       # step 5
frame = Frame(plt.root)                            # step 6
graph = plt.makeGraphBase(frame, 500, 300,
                          title="Line and bars")   # step 7
graph.pack()                                       # step 8
graph.draw(obj)                                    # step 9
frame.pack()                                       # step 10
graph.postscr()                                    # step 11
plt.mainloop()                                     # step 12
```

Which generates:



Clearly, this level API is more verbose, but allows putting several diagrams with different plot types into one plot, or putting putting several plots into one frame (side by side, vertically, or in table fashion).

title

Assign a title to appear in the container's title bar. (This is a method exposed by a Tk Toplevel container.)

Call: <rootInstance>.title(title)

Mandatory parameters:

- **title** : the title to appear in the container's title bar (string)

Optional parameters: None.

Return value: None.

makeLine

Generates a line plot object from a list of tuples (or lists).

Call: <variable> = <SimPlotInstance>.makeLine(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples)

Optional parameters:

- **color = 'black'** : line colour (Tk colour value)
- **width = 1** : line width (integer or float)
- **smooth = False** : smooth line if True (boolean)
- **splinesteps = 12** : number of spline steps for smoothing line (integer); the higher, the better the line follows the points provided

Return value: Reference to a line plot object (GraphLine)

makeStep

Generates a line plot object from a list of tuples (or lists). A horizontal line is generated at a y-value until y changes, creating a step effect.

Call: <variable> = <SimPlotInstance>.makeStep(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples)

Optional parameters:

- **color = 'black'** : line colour (Tk colour value)
- **width = 1** : line width (integer or float)

Return value: Reference to a line plot object (GraphLine)

makeSymbols

Generates a scatter diagram plot object with markers from a list of tuples (or lists).

Call: <variable> = <SimPlotInstance>.makeSymbols(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples)

Optional parameters:

- **marker='circle'** : symbol type (literal; values supported: 'circle, 'dot', 'square', 'triangle, 'triangle_down', 'cross', 'plus')
- **width=1** : line drawing width (integer or floating point)
- **color='black'** : line colour (Tkinter colour type; see section on Colours in SimPlot)
- **fillcolor='black'** : colour with which bars are filled (Tkinter colour type; see section on Colours in SimPlot)
- **fillstyle='‘** : density of fill (default=100%; Tkinter bitmap)
- **outline='black'** : colour of marker outline ((Tkinter colour type; see section on Colours in SimPlot)

Return value: Reference to a scatter plot object (GraphSymbols)

makeBars

Generates a bar chart plot object with markers from a list of tuples (or lists).

Call: <variable> = <SimPlotInstance>.makeBars(values[,optional parameters])

Mandatory parameters:

- **values** : a list of two-element lists (or tuples)

Optional parameters:

- **width=1** : width of bars (integer or floating point)
- **color='black'** : bar colour (Tkinter colour type; see section on Colours in SimPlot)
- **fillcolor='black'** : colour with which bars are filled (Tkinter colour type; see section on Colours in SimPlot)

- **fillstyle='** : density of fill (default=100%; Tkinter bitmap)
- **outline='black"** : colour of bar outline ((Tkinter colour type; see section on Colours in SimPlot)

Return value: Reference to a bar chart plot object (GraphSymbols)

makeHistogram

Generates a histogram plot from a Histogram or a Histogram-like list or tuple. A SimPy Histogram instance is a list with items of two elements. It has n+2 bins of equal width, sorted by the first element, containing integer values == the counts of the bins. The first bin is the 'under' bin, the last the 'over' bin. Histogram objects are produced from Monitor objects by calling the Monitor method *histogram()*.

Call: <variable> = <SimPlotInstance>.makeBars(values[,optional parameters])

Mandatory parameters:

- **values** : a Histogram-like object

Optional parameters:

- **width=1** : width of line (integer or floating point)
- **color='black'** : line colour (Tkinter colour type; see section on Colours in SimPlot)

makeGraphObjects

Combines one or mor plot objects into one plottable GraphObject.

Call: <variable> = <SimPlotInstance>.makeGraphObjects(list_of_plotObjects)

Mandatory parameters:

- **list_of_plotObjects** : a list of plot objects

Optional parameters: None

Return value: Reference to a GraphObject

makeGraphBase

Generates a canvas widget in its Tk container widget (such as a Frame) with the plot's background (title, axes, axis labels).

Call: <variable> = <SimPlotInstance>.makeGraphBase(master, width, height [,optional parameters])

Mandatory parameters:

- **master** : container widget for graph widget
- **width** : width of graph widget in pixels (positive integer)
- **height** : height of graph widget in pixels (positive integer)

Optional parameters:

- **background='white'** : colour of plot background (Tk colour value)
- **title="** : title of plot (string)
- **xtitle='** : label on x-axis (string)
- **ytitle='** : label on y-axis (string)

Return value: Reference to a GraphBase object (graph widget)

pack

Controls how graph widget is arranged in its master container. (Inherited from Tk Packer geometry manager.)

Call: <GraphBaseInstance>.pack([optional parameters])

Mandatory parameters: None.

Optional parameters:

- **side** : where to place graph widget (side=LEFT: to the left; side=TOP: at the top; Tk Packer literals)
- **fill** : controls whether graph fills available space in window (fill=BOTH: fills in both directions; fill=X: horizontal stretching; fill=Y: vertical stretching)
- **expand=NO** : controls whether Packer expands graph widget when window is resized (expand=TRUE: widget may expand to fill available space)

Return value: None

draw

Draws the plot background and the lines/curves in it.

Call:

<GraphBaseInstance>.draw(graph,[optional parameters])

Mandatory parameters:

- **graphics** : graph widget (GraphBase) instance

Optional parameters:

- **xaxis='automatic'** : controls appearance of x-axis (None: no x-axis; "minimal": axis runs exactly from minimal to maximal x-value; "automatic" : x-axis starts at 0 and includes maximal x-value; tuple (xlow,xhigh) = have x-axis go from xlow to xhigh)
- **yaxis='automatic'** : controls appearance of y-axis (None: no y-axis;"minimal": axis runs exactly from minimal to maximal y-value; "automatic" : y-axis starts at 0 and includes maximal y-value; tuple (ylow,yhigh) = have y-axis go from ylow to yhigh

Return value: None

postscr

After call to draw , saves Postscript output from a plot to a file.

Call: <GraphBaseInstance>.postscr([optional parameter])

Mandatory parameters: None.

Optional parameters with defaults:

- **"filename"** : name of file (complete path) to which Postscript output is written. If omitted, a dialog asking the user for a filename pops up.

Return value: None.

5.1.5 Colours in SimPlot

Colours in SimPlot are defined by Tk. The Tk colour model is RGB. The simplest way to identify a colour is to use one of the hundreds of Tk-defined literals such as "black", "aquamarine", or even "BlanchedAlmond". See the Tk colour page for definitions.

5.2 SimGUI Manual

Authors Klaus Muller <Muller@users.sourceforge.net>

SimPy Classic Release 2.3.3

Web-site https://github.com/SimPyClassic/SimPyClassic

Python-Version 2.7 and later

Date December 2011

Updated January 2018

Contents

This manual describes **SimGUI**, a GUI framework for SimPy simulation applications.

5.2.1 Acknowledgements

The initial ideas for using a Tk/Tkinter-based GUI for SimPy simulation models and first applications came from Mike Mellor and Prof. Simon Frost of University of California, San Diego. Simon has been a very productive co-developer of SimGUI.

Following an idea by Simon Frost, SimGUI uses a great Tkinter-based console for conversing with the Python interpreter, developed by Ka-Ping Yee <ping@lfw.org>.

5.2.2 Introduction

SimGUI is a GUI framework for SimPy simulation programs. It provides for:

- a standard layout of the user interface, including its menu structure
- running a simulation
- viewing simulation output
- saving the output to a file

- changing the simulation model parameters
- viewing a model description
- viewing the simulation program's code
- a Python console for debugging

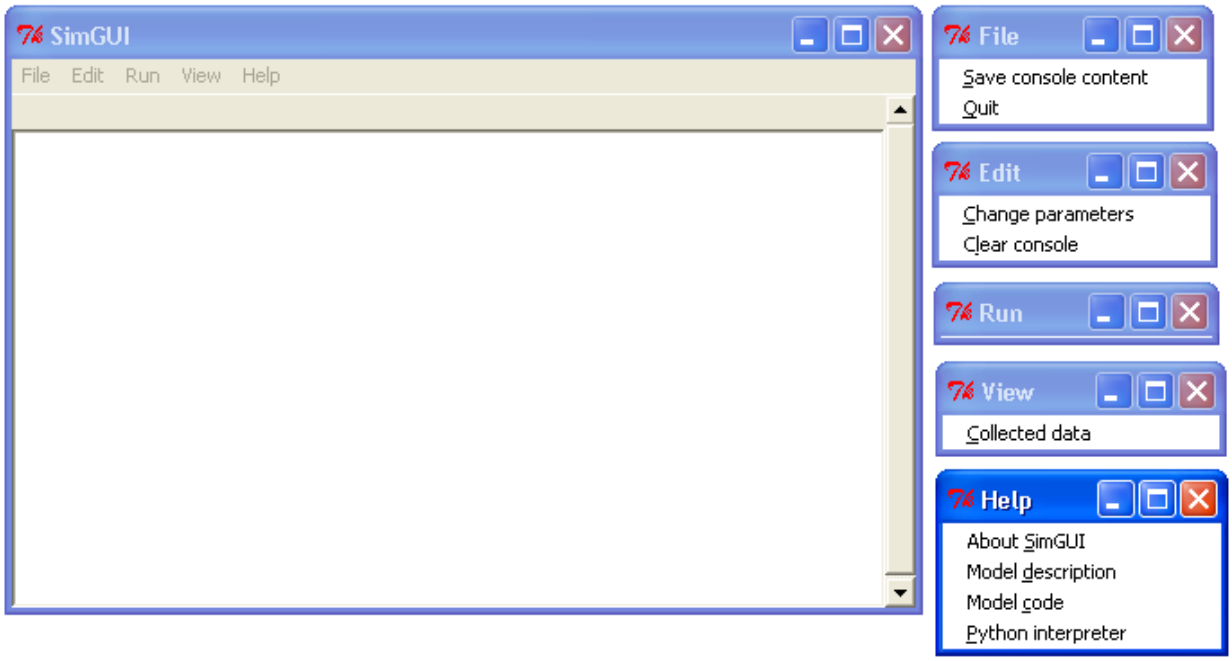
SimGUI is based on the Tk GUI library.

5.2.3 SimGUI overview

Here is a minimalistic program which does nothing but show the SimGUI user interface:

```
## SimGUIminimal.py
from SimPy.SimGUI import *
root=Tk()
gu=SimGUI(root,consoleHeight=20)
gu.mainloop()
```

Running it produces this output:



The large frame is the **SimGUI application window**. To its right are the standard menu items of SimGUI. To show them, the menus have been torn off by clicking on the dotted line on all SimGUI drop-down menu items.

The SimGUI application window consists of five widgets:

- the outside frame is a Tk Toplevel widget with a default title (which can be changed),
  - a **menu bar**, a Tk Menu widget which can be adapted by the application program (contained in toplevel)
  - the **output window**, a Tk Frame widet for SimPy program output (contained in toplevel)
    - the **status bar** for one-line status messages, a Tk Label widget (contained in output window)
    - the **output console** for application output, a Tk Text widget (contained in output window)

The **File** sub-menu is for saving or opening files and also for quitting the application. By default, it supports saving the content of the output console.

The **Edit** sub-menu is for any editing or change operations. By default, it supports changing model parameters and clearing the output console.

The **Run** sub-menu is for running the SimPy model. By default, it is empty, as there is no model to run.

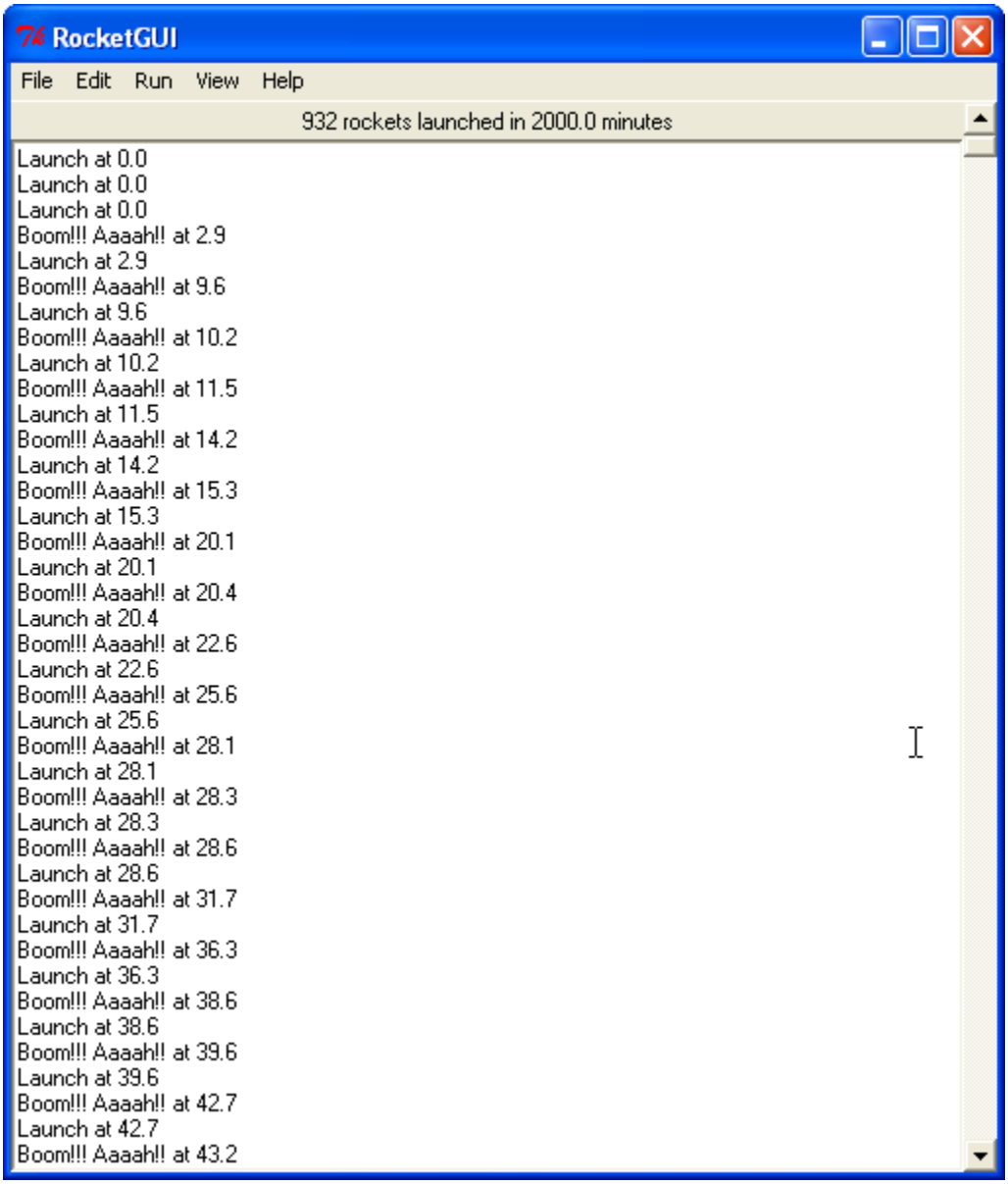
The **View** sub-menu is for calling up any application output, e.g. plots or statistics. By default, it has the ability to automatically show the data from any Monitor instance in the SimPy application.

The **Help** sub-menu is for viewing any information about SimGUI and the specific application behind it. By default, it provides information about the version of SimGUI, lists the application's code and allows the launching of a Python console for testing and debugging purposes.

To show how simple it is to interface a model to SimGUI, here is an example with a simple simulation model:

```
#!/usr/bin/env python
__doc__="""GUIdemo.py
This is a very basic model, demonstrating the ease
of interfacing to SimGUI.
"""
from __future__ import generators
from SimPy.Simulation  import *
from random import *
from SimPy.SimGUI import *

class Launcher(Process):
    nrLaunched=0
    def launch(self):
        while True:
            gui.writeConsole("Launch at %.1f"%now())
            Launcher.nrLaunched+=1
            gui.launchmonitor.observe(Launcher.nrLaunched)
            yield hold,self,uniform(1,gui.params.maxFlightTime)
            gui.writeConsole("Boom!!! Aaaah!! at %.1f"%now())

def model():
    gui.launchmonitor=Monitor(name="Rocket counter",
                              ylab="nr launched",tlab="time")
    initialize()
    Launcher.nrLaunched=0
    for i in range(gui.params.nrLaunchers):
        lau=Launcher()
        activate(lau,lau.launch())
    simulate(until=gui.params.duration)
    gui.noRunYet=False
    gui.writeStatusLine("%s rockets launched in %.1f minutes"%
                        (Launcher.nrLaunched,now()))

class MyGUI(SimGUI):
    def __init__(self,win,**par):
        SimGUI.__init__(self,win,**par)
        self.run.add_command(label="Start fireworks",
                             command=model,underline=0)
        self.params=Parameters(duration=2000,maxFlightTime=11.7,nrLaunchers=3)

root=Tk()
gui=MyGUI(root,title="RocketGUI",doc=__doc__,consoleHeight=40)
gui.mainloop()
```

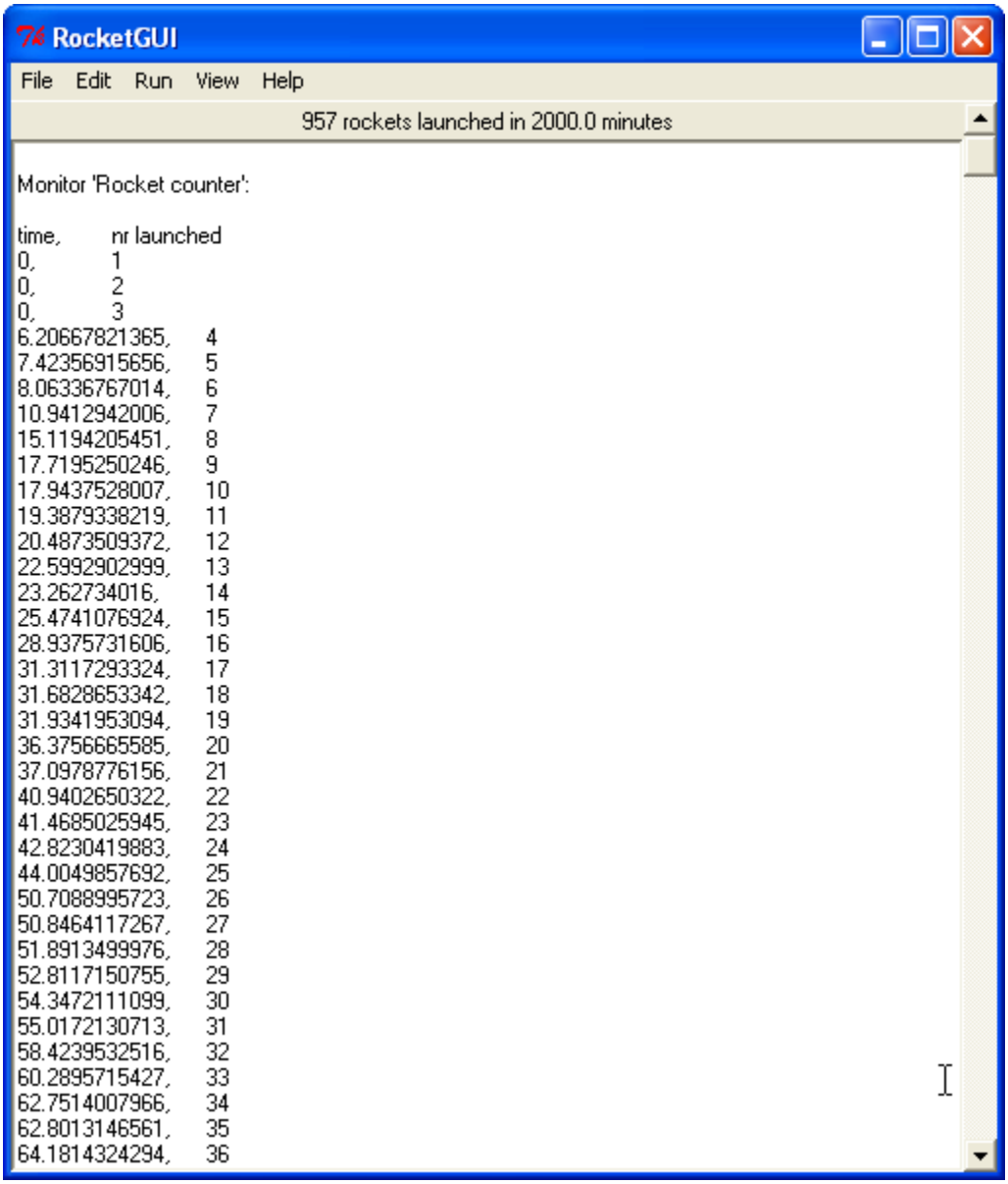
It produces the following output when the model run command is selected:



class MyGUI adds one menu item under the Run menu. It also defines three parameters (duration, maxFlighTime and nrLaunchers) with their initial values which the user can change interactively before each run. The MyGUI instance named gui sets the window title, the model description (the __doc__ string) and the height of the output window.

The simulation part of the program writes to the output console and to the status line.

The model uses a Monitor for keeping track of the number of rockets launched over time. Because it is made an attribute of the MyGUI instance, the super class (SimGUI) can output the Monitor after a run. This requires no application code. When the menu item 'Collected data' under the View menu is selected, this results in:



5.2.4 The SimGUI API

Structure

The SimGUI module exposes the following API to the applications programmer:

```
class SimGUI(object)
        def __init__
                self.doc = doc
                self.noRunYet=True
                self.top = Menu
                self.file = Menu
                self.edit = Menu
                self.run = Menu
                self.view = Menu
```

```
                self.help = Menu
        def makeFileMenu
        def makeEditMenu
        def makeRunMenu
        def makeViewMenu
        def makeHelpMenu
        def writeConsole
        def saveConsole
        def clearConsole
        def writeStatusLine
        def about
        def notdone
        def showTextBox
        def mainloop

class Parameters
        def __init__
        def show
```

class SimGUI, __init__ constructor

Encapsulates the SimGUI functionality.

Call: <variable>.SimGUI(win[,optional parameters])

Mandatory parameters:

- **win** : the master widget in which the SimGUI widgets are embedded

Optional parameters:

- **title=SimGUI** : the title of the top level window (string)
- **doc=None** : the model description (string)
- **consoleHeight=50** : the height of the output console in lines (positive integer)

Return value: Returns a reference to the SimGUI instance.

SimGUI instance fields

In addition to the constructor parameters, the SimGUI fields of interest to the applications programmer are:

- **self.noRunYet=True** : a predicate indicating whether the model has been run yet; must be set to True after each model run; should be tested by application program before any run-dependent output is produced (boolean)
- **self.top = Menu** : the top level menu widget (menu bar)
- **self.file = Menu** : the 'File' menu widger
- **self.edit = Menu** : the 'Edit' menu widget
- **self.run = Menu** : the 'Run' menu widget
- **self.view = Menu** : the 'View' menu widget
- **self.help = Menu** : the 'Help' menu widget

Adding menu items

Menu items can be added to SimGUI submenus by:

<menufield>.add_command(label= <string,command=<callable>, underline=<integer>)

E.g. *self.run.add_command(label="Start fireworks",command=model,underline=0)*. This is all standard Tk/Tkinter syntax – read any Tk/Tkinter manual or book.

Changing menus

Any of the submenus provided by SimGUI can be replaced by overloading one or more of the methods **makeFileMenu, makeEditMenu, makeRunMenu, makeViewMenu, makeHelpMenu**. This is done by defining the methods to be overloaded in the SimGUI subclass.

The overloading method should look like:

```
def makeFileMenu():
        self.file = Menu(self.top)
        self.file.add_command(label='Save some results',
                      command=self.saveResults, underline=0)
        self.file.add_command(label='Get out of here',
                 command=self.win.quit,underline=0)
        self.top.add_cascade(label='File',menu=self.file,underline=0)
```

Note: It is advisable to keep the basic the SimGUI menu structure in order to maintain the SimGUI look-and-feel.

writeConsole

Writes a string into the output console Text widget, with newline at end.

Call:

<SimGUI instance>.writeConsole(text)

Mandatory parameters: None.

Optional parameters:

- **text='‘** : text to write (string)

Return value: None.

saveConsole

Saves output console to a text file which the user is prompted to name.

Call: <SimGUI instance>.saveConsole()

Mandatory parameters: None

Optional parameters: None

Return value: None

clearConsole

Clears output console.

Call: <SimGUI instance>.clearConsole()

Mandatory parameters: None

Optional parameters: None

Return value: None

writeStatusLine

Writes a one-line string to the status line.

Call:

<SimGUI instance>.writeStatusLine(text)

Mandatory parameters: None.

Optional parameters:

- **text=''** : text to write (string, not longer than 80 character)

Return value: None.

notdone

Brings up a warning box staing that a feature is not implemented yet. Useful during development of application.

Call:

<SimGUI instance>.notdone()

Mandatory parameters: None.

Optional parameters: None.

Return value: None.

showTextBox

Pops up a text box (Text widget)with a text in it.

Call:

<SimGUI instance>.showTextBox(optional parameters)

Mandatory parameters: None.

Optional parameters:

- **title=''** : title of text box (string)
- **text=''** : text to write (string)
- **width=80** : width of box in characters (positive integer)
- **height=10** : height of box in lines (positive integer)

Return value: None.

mainloop

Starts up SimGUI execution.

Call: <SimGUI instance>.mainloop()

Mandatory parameters: None.

Optional parameters: None.

Return value: None

class Parameters, __init__ constructor and adding parameters

This class provides for interactive user changes of model parameters. Any user-input is checked against the type of the original (default) value of the parameter.In this version, parameters of type integer, floating point, text and list are supported. Boolean parameters can be implemented by using 0 for False and 1 for True.

Example: gui.params=Parameters(endtime=2000, numberCustomers=50, interval=10.0, trace=0)

This results in parameters gui.params.numberCustomers, gui.params.interval and gui.params.trace.

Call: <SimGUI instance>.params=Parameters(par) (constructor) **<SimGUI instance>.params.<name>=<value>** (adding parameters)

Mandatory parameters:

- **par** : one or more pairs <*name*>=<*value*>, separated by commas. <value> must be one of the four types supported.
- **<name>** : the parameter name (any legal Python identifier)
- **<value>** : the parameter's initial value (must be one of the four types supported, i.e. integer/boolean, floating point, text and list)

Optional parameters: None.

Return value: A Parameter instance.

show

Returns the parameter name/value pairs of a Parameter instance in pretty-printed form (one pair per line).

Call: <Parameter instance>.show()

Mandatory parameters: None.

Optional parameters: None.

Return value: A string with the name/value pairs separated by newline ('n').

5.2.5 Miscellaneous SimGUI capabilities

Source code listing

The 'Model code' item of the 'Help' submenu lists the application code of a running SimGUI application by outputting the content of sys.argv[0]. No user programming is required for this.

Automatic display of Monitor instances

After a model run, any Monitor instance which is referred to by a SimGUI (sub)class instance is shown in the output console by the 'Collected data' item on the 'View' submenu. Just e.g. **gui.waitMon=Monitor("Waiting times")**, where gui refers to a SimGUI (sub)class instance, is enough to facilitate this.

$Revision: 297 $ $Date: 2009-03-31 02:24:46 +1300 (Tue, 31 Mar 2009) $ kgm

5.3 Visualize model events and states using SimulationGUIDebug

Brian Jacobs, Kip Nicol and Logan Rockmore, a group of senior students of Professor Matloff at U. of California at Davis have developed a great tool for gaining insight into the event-by-event execution of a SimPy model. It is equally useful for teaching and program debugging.

The **SimulationGUIDebug** tool is a SimPy module which shows a GUI with a model's event list and the status of selected Process and Resource instances over time. The user can interactively step from event to event and set breakpoints at one or more points in (simulated) time.

SimulationGUIDEbug can be used as a standalone debuger or together with Python debuggers such as PDB.

The tool was originally written for SimPy 1.9. It has been ported to SimPy 2.0.

The tool's documentation (user guide, design overview, etc.) is here .

CHAPTER 6

Acknowledgments

SimPy 2.3.3 has been primarily developed by Stefan Scherfke and Ontje Lünsdorf, starting from SimPy 1.9. Their work has resulted in a most elegant combination of an object oriented API with the existing API, maintaining full backward compatibility. It has been quite easy to integrate their product into the existing SimPy code and documentation environment.

Thanks, guys, for this great job! **SimPy 2.0 is dedicated to you!**

The many contributions of the SimPy user and developer communities are of course also gratefully acknowledged.

CHAPTER 7

Indices and tables

- genindex
- search

Index

P

Q

R