Simulation Programming using PythonSim

Vahid Sarhangian
University of Toronto

Adapted from Slides by Barry Nelson

Northwestern University

July 2017

Discrete-event simulation

- The foundations of DES modeling & programming:
  - **System State**: Variables and data structures that hold all needed information about the system.
  - **Events**: Countable points in time that change the state in well-defined ways & schedule future events.
  - **Clock**: Current time in the simulation
  - **Event Calendar**: List of future events yet to be executed in chronological order.
- We want to learn programming approaches and structures that facilitate developing large-scale simulation models.

Object orientation

- Discrete-event simulations often contain **multiple instances of similar objects**:
  - **Entities**: things that come and go, like customers, messages, jobs, events
  - **Queues**: ordered lists of entities
  - **Resources**: scarce quantities that entities need, like servers, computers, machines
  - **Statistics**: information to be recorded on all of the above
- These are more naturally treated as “objects” for which we can have many instances.

PythonSim

- A module containing Python classes (objects) that support simulation.
  - You can easily customize and add to these
- A module containing a few useful functions.
- Implementations of the random-number and random-variate generation functions from `simlib` by Law & Kelton.
- Other packages, e.g., SimPy

Python classes

- With classes we define the template for an object.
- A **class** contains…
  - **attributes** → data values
  - **methods** → instructions about how to do things (functions)
- The key benefit of objects is that we can create multiple instances, each uniquely identified and with its own attributes and methods.

CTStat class

```
class CTStat():
    def __init__(self):
        self.Area = 0.0
        self.Tlast = 0.0
        self.TClear = 0.0
        self.Xlast = 0.0

    def Record(self,X):
        self.Area = self.Area + self.Xlast * (Clock - self.Tlast)
        self.Tlast = Clock
        self.Xlast = X

    def Mean(self):
        mean = 0.0
        if (Clock - self.TClear) > 0.0:
            mean = (self.Area + self.Xlast*(Clock - self.Tlast))/(Clock - self.TClear)
        return mean

    def Clear(self):
        self.Area = 0.0
        self.Tlast = Clock
        self.TClear = Clock
```

← Automatically called when a New CTStat is created

← Attributes: each CTStat will have its own copy

← Called from within your simulation to update the statistic

← Called from within your simulation to report the sample mean

← Called from within your simulation to reset the CTStat

Anatomy of a class

- Declarations
  - Defines the name of the class (e.g., CTStat)
  - Defines `__init__(self):` a method called when a new instance is created
    - Can define the attributes and their initial values
    - Each instance of the object will have its own unique copy
  - Think of `self` as a pointer to the specific instance; this is how we can have multiple unique instances

```
class CTStat():
    def __init__(self):
        self.Area = 0.0
        self.Tlast = 0.0
        self.TClear = 0.0
        self.Xlast = 0.0
```

Anatomy of a class

- Methods (functions) take arguments and return results just like regular functions
- `self` must be the first argument and used to reference attributes via `self.attributename`

```
def Record(self,X):
    self.Area = self.Area + self.Xlast * (Clock - self.Tlast)
    self.Tlast = Clock
    self.Xlast = X

def Mean(self):
    mean = 0.0
    if (Clock - self.TClear) > 0.0:
       mean = (self.Area + self.Xlast*(Clock - self.Tlast))/(Clock - self.TClear)
    return mean

def Clear(self):
    self.Area = 0.0
    self.Tlast = Clock
    self.TClear = Clock
```

Using PythonSim

- `SimFunctions.py`
  - Functions `SimFunctionsInit,Schedule,SchedulePlus,ClearStats`
- `SimClasses.py`
  - `CTStat, DTStat`
  - `Entity`
  - `EventCalendar, EventNotice`
  - `FIFOQueue`
  - `Resource`
- `SimRNG.py`
  - Random-number generator and a few common distributions.

Python Lists

- A **List** is a Python generalization of an array:
  - It is **ordered** and can store anything, including class objects.
  - It is dynamic, increasing and decreasing in size as we add and remove.
- We use lists for **event calendar** management and **queue** management.

List syntax

`Queue=[]` ← declare the list

`Queue[0]` ← element in the first position

`Queue[-1]` ← element in the last position

`Queue[3]` ← element in 3 position (fourth)

`Queue.remove(value)` ← remove element with this value

`Queue.insert(7, whatever)` ← insert element in the 7 position (eighth)

`Queue.append(whatever)` ← insert element in the last position

Model of an M/G/1 queue

- Single-server queue with exponential time between arrivals, and Erlang service time.
- Key objects:
  - A `Resource` called `Server`
  - A `FIFOQueue` for customers called `Queue`
  - A discrete-time statistic (`DTStat`) to collect total time in the system called `Wait`
  - An `EventCalendar` called `Calendar`
- We make 10 replications of 55,000 time units, deleting the first 5000.

The basic logic

- Customer arrival
  - Schedule the next arrival
  - Put customer in the queue
  - If the server is idle, make busy and schedule an end of service.
- End of service
  - Remove the customer from the queue & record their time in system
  - If more customers in queue, start next service; Else make the server idle

Using SimRNG

- `SimRNG.InitializeRNSeed()`
  - Call **once** at the beginning of the simulation to initialize the pseudorandom-number generator
- `SimRNG.lcgrand(Stream)`
  - Pseudorandom-number generator
  - Streams 1-100
- `SimRNG.[Expon, Erlang, Random_integer, Normal, Lognormal, Triangular]`
  - Arguments are distribution parameters first, with last argument being the stream number 1-100
  - Ex: `SimRNG.Expon(15.2, 7)`

M/G/1 Queue in PythonSim

```
import SimFunctions
import SimRNG
import SimClasses
import pandas
Import numpy as np
```

All simulations need these 3 modules

```
ZSimRNG = SimRNG.InitializeRNSeed()
```

Initialize random-number generator outside loop

```
Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()
```

Create instances of the needed class objects

```
TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []
```

Objects in these lists will be reinitialized between replications

```
TheDTStats.append(Wait)
TheQueues.append(Queue)
TheResources.append(Server)
```

Add the objects to the appropriate lists

```
Server.SetUnits (1)
MeanTBA = 1.0
MeanST = 0.8
Phases = 3
RunLength = 55000.0
WarmUp = 5000.0
```

The SetUnits method sets the number of units of the Resource available.

```
AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
```

These are just lists to save the output results; not required

SimClasses.Clock initiated at 0 and globally updated

```
def Arrival():
    SimFunctions.Schedule(Calendar,"Arrival",SimRNG.Expon(MeanTBA, 1))
    Customer = SimClasses.Entity()
    Queue.Add(Customer)

    if Server.Busy == 0:
        Server.Seize(1)
        SimFunctions.Schedule(Calendar,"EndOfService",
                              SimRNG.Erlang(Phases,MeanST,2))



def EndOfService():
    DepartingCustomer = Queue.Remove()
    Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)

    if Queue.NumQueue() > 0:
        SimFunctions.Schedule(Calendar,"EndOfService",
                              SimRNG.Erlang(Phases,MeanST,2))
    else:
        Server.Free(1)
```

- SimFunctionsInit clears out the event calendar, statistics, queues and resources between replications
- We always need to schedule one or more initial events
- We then find the first event and get the simulation started

```
for reps in range(0,10,1):
    SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,
    TheDTStats,TheResources)
    SimFunctions.Schedule(Calendar,"Arrival",SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar,"EndSimulation",RunLength)
    SimFunctions.Schedule(Calendar,"ClearIt",WarmUp)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Arrival":
        Arrival()
    elif NextEvent.EventType == "EndOfService":
        EndOfService()
    elif NextEvent.EventType == "ClearIt":
        SimFunctions.ClearStats(TheCTStats,TheDTStats)
```

```
while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Arrival":
            Arrival()
        elif NextEvent.EventType == "EndOfService":
            EndOfService()
        elif NextEvent.EventType == "ClearIt":
            SimFunctions.ClearStats(TheCTStats,TheDTStats)

print (reps+1, Wait.Mean(), Queue.Mean(),
               Queue.NumQueue(), Server.Mean()
```

The basic simulation loop ending at a particular event type

Here we use the built-in statistics collection of FIFOQueue and Resource, plus the DTStat

PythonSim program structure

```
import SimFunctions
import SimRNG
import SimClasses

ZSimRNG = SimRNG.InitializeRNSeed()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

def SomeEvent():

for reps in range(0,10,1):
    SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,
         TheDTStats,TheResources)

SimFunctions.Schedule(Calendar, "SomeEvent", SomeTime)

    while SomeConditionIsTrue:
        NextEvent = Calendar.Remove()
        Clock = NextEvent.EventTime
        If NextEvent.EventType == "SomeEvent"
           SomeEvent()

    # Collect any within statistics from rep

# Report any overall statistics from simulation
```

Initialize random-number generator and create the event calendar

Create and fill the class objects to be cleared between replications

Build your event functions

Reset the class objects for each new replication

Schedule some events

The basic simulation loop

Using PythonSim

- `SimFunctions.py`
  - Functions `SimFunctionsInit,Schedule,SchedulePlus,ClearStsts`
- `SimClasses.py`
  - `CTStat, DTStat`
  - `Entity`
  - `EventCalendar, EventNotice`
  - `FIFOQueue`
  - `Resource`
- `SimRNG.py`
  - Random-number generator and a few common distributions.

SimFunctionsInit

- Usage:
  `SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,TheDTStats,TheResources)`
- Typically placed **inside** the replication loop
- Resets `Calendar`, and all of `TheXXX[…]` lists.
- The order of arguments matters

Schedule & SchedulePlus

```
SimFunctions.Schedule(calendar, EventType, EventTime)

SimFunctions.SchedulePlus(Calendar, EventType, EventTime,
TheObject)
```

- Usage:

```
SimFunctions.Schedule(Calendar, "Arrival", RNG.Expon(2,1))

NewCustomer = SimClasses.Entity()
SimFunctions.SchedulePlus(Calendar, "Arrival", RNG.Expon(2,1),
       NewCustomer)
```

- Notice that EventTime is how far into the future the event is to occur, not the absolute time.
- The "Plus" version allows another object (usually an Entity) to be attached to the EventNotice.
- EventType should be a string variable.

Entity class

- Usage

```
NewCustomer = SimClasses.Entity()

Delay = SimClasses.Clock - NewCustomer.CreateTime
```

- You can add as many additional attributes as you need to the Entity Class.

```
class Entity():
    CreateTime = 0.0
    def __init__(self):
        self.CreateTime = Clock
#   add other attributes here...
```

EventNotice class

- Usage

```
NextEvent = Calendar.Remove()

SimClassesClock = NextEvent.EventTime
If NextEvent.EventType == "SomeEvent"

     Call SomeEvent(NextEvent.WhichObject)
```

If using SchedulePlus

- The EventNotices are usually created by Schedule or SchedulePlus; you use them when advancing to the next event.

```
class EventNotice():
    def __init__(self):
        self.EventTime = 0.0
        self.EventType = ""
        self.WhichObject = ()
```

CTStat class

- Collects continuous-time statistics
- Methods: Record, Mean and Clear
- Usage

```
InvtLevelStat = SimClasses.CTStat()

InvtLevelStat.Record(InvtLevel)

InvtLevelStats.Mean()

InvtLevelStats.Clear()
```


(Time-average up to time Clock)

DTStat class

- Collects discrete-time statistics
- Methods: Record, Mean, StdDev, N and Clear
- Usage

```
TimeInSystemStat = SimClasses.DTStat()

TimeInSystemStat.Record(SimClasses.Clock -
Customer.CreateTime)

TimeInSystemStat.Mean()
TimeInSystemStat.StdDev()
TimeInSystemStat.N()

TimeInSystemStat.Clear()
```

Resource class

- Models resources and also keeps a CTStat on average number in use
- Attributes: Busy [current number in use]
- Methods: SetUnits, Seize, Free, Mean
- Usage

```
Worker = SimClasses.Resource()

Worker.SetUnits(5)        # resource has capacity 5
Worker.Seize(1)           # make 1 unit of resource busy
Worker.Free(1)            # make 1 unit of resource idle

If Worker.Busy == 5 Then ...

Worker.Mean()             # average number in use
```

FIFOQueue class

- Models first-in-first-out queue, and also keeps a CTStat on number in queue
- Methods: NumQueue, Add, Remove, Mean
- Usage

```
MyLine = SimClasses.FIFOQueue()

Shopper = SimClasses.Entity()
MyLine.Add(Shopper)

DepartingShopper = MyLine.Remove()

If MyLine.NumQueue() = 0 Then ...

MyLine.Mean()     # average number in queue
```

Some notes…

- The CTStat's created by `FIFOQueue` and `Resource` are automatically added to `TheCTStats` collection, so they are reinitialized by `SimFunctionsInit`.
- The most common change you will make is to add attributes to the Entity class.
- When referencing a method there is a `()` even if no arguments, while an attribute has none:

  `MyLine.NumQueue()` ← because it is computed

  `Server.Busy`