CSC148 homework#4 (h4) Prof Mitchell 10/11/19 Due: 19/11 Revision: Nov 15, 11am (See Appendix & RED markup)

This first simpy3 (simpy) assignment implements a part of order/inventory (aka oi) processing that is commonplace in almost all commerce.

Conceptually, oi is different from customer service models because cj orders reduce a site’s inventory; unlike service, reduced inventory is consumed and replaced.

You are to code a simpy script that simulates order/delivery activity between a single large warehouse named W and three local Home Depot stores named HD1, HD2, and HD3. The DES time unit is 1 hour. The HDj are 24X7 operations, and each HDj is initially stocked with 200 roofing shingle bags. All HDj open and start business at, say 6AM, on a Monday.

For simplicity, there is only one kind of customer order: n 90-pound bags of roofing shingles. Sales orders are at the “Pro desk” in each HDj. Successive customer (general construction contractor) orders occur 1 or 2 hours apart (uniformly distributed). Each order size n is a uniformly random int in ~~[6,35]~~ [10,45]. (*See Appendix: numpy distributions*)

The model – part of a simple inventory system

Each HD processes orders, one-by-one, and each order decrements an HD’s remaining inventory by n. However, order processing terminates each HD as soon as a size n order decrements HD’s inventory below level RP = 50 (RP stands for “reorder point”). And when ~~all~~ each HD’s inventory level has dropped below RP, that HD finishes. And, the model run terminates after all HD have finished.

You are required to implement W and HD as separate classes. ~~Although we will not expand this model beyond one W and three HDs,~~ Two classes cleanly separate W and HD behaviors.

(*A model expansion could have any number of W, each W having its own group of HD stores*)

Component sections of a simpy script

Create Python and simpy source code in the idle command window (not your favorite text editor). idle does a lot of automatic (and required) tab placement for statement blocking. The order of sections of a typical simpy script is described next.

imports

At the top of the script, import simpy, random (for internal randomness calculations),

numpy as np (for numerical processing, and specifically to access well-known probability pmfs), and pdb (this is optional, only if needed for debugging).

“import numpy as np” illustrates how to abbreviate a long module name, so that numpy.x can be coded np.x. *Note – technically, it is named NumPy, but we refer to NumPy as simply numpy*

Seeding

Similar with gpss seeding, code (but, as always, document each statement)

seedName = someMultiDigitInteger

np.random.seed(seedName)

Class layout and conventions

Class Classname**:** *< -- A class name is capitalized; class parameters probably not needed in CSC148*

Docstring text documentation (consisting of >= 1 text line) surrounded by triple double quotes

def \_\_init\_\_(self,env, a list of parameters if needed)**:** *< -- User-defined classes have an \_\_init\_\_() function (constructor)*

Any number of statements specifying class initializations *(properly indented as function body statements)*

*(For CSC148, there is typically no need for a return statement in most functions)*

*The \_\_init\_\_ is followed by >=0 function definitions (each function can be either a pf or an ordinary Python function)*

def fcnName(self,env, a list of parameters if needed)**:**

Any number of statements *(properly indented as function body statements)*

A class definition ends with end of source code indentation of class items

Using numpy for calls to distribution pmfs

The expression “np.random.uniform(x,y)” returns a numeric value in the range [x,). Its value can be assigned to a variable. *Note the interval endpoint inclusion and exclusion.*

Required HandIns

1. A complete copy of your source code
2. Complete output (*with exact display shown below*) for 2 model runs with different numpy seeds

Sample model run results

=========== RESTART: C:/Users/bill/148\_s19/homework/h4\_148\_f19.py ===========

Running class Warehouse \_\_init\_\_ fcn at time 0 for warehouse WH1 // NumPy seed is 680771

Creating 3 HD stores

Running HD 1 \_\_init\_\_fcn at time 0 Setting Inventory & RP values to 200 & 50

Running HD 2 \_\_init\_\_fcn at time 0 Setting Inventory & RP values to 200 & 50

Running HD 3 \_\_init\_\_fcn at time 0 Setting Inventory & RP values to 200 & 50

Starting doOrders() for HD 1 at time 0

Starting doOrders() for HD 2 at time 0

Starting doOrders() for HD 3 at time 0

HD 1 inventory level is 174 at time 1

HD 3 inventory level is 185 at time 1

HD 2 inventory level is 164 at time 2

HD 1 inventory level is 134 at time 3

HD 3 inventory level is 144 at time 3

HD 2 inventory level is 150 at time 3

HD 1 inventory level is 90 at time 4

HD 3 inventory level is 123 at time 4

HD 2 inventory level is 135 at time 4

HD 1 inventory level is 53 at time 5

HD 2 inventory level is 124 at time 5

HD 3 inventory level is 81 at time 6

HD 1 inventory level is 38 at time 7

Finished HD 1 Ops at time 7 with inventory 38

HD 2 inventory level is 98 at time 7

HD 3 inventory level is 43 at time 8

Finished HD 3 Ops at time 8 with inventory 43

HD 2 inventory level is 54 at time 9

HD 2 inventory level is 38 at time 11

Finished HD 2 Ops at time 11 with inventory 38

Finished run at model time 11

>>>

Appendix

**Calling distribution pmfs in numpy**

Source: **NumPy v1.14 Manual**

numpy.random.uniform**(***low=0.0***,***high=1.0***,***size=None***)**

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by [**uniform**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.uniform.html#numpy.random.uniform).

|  |  |
| --- | --- |
| **Parameters:** | **low***: float or array\_like of floats, optional*  Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.  **high***: float or array\_like of floats*  Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0. |

**Class and function parameters in simpy**

Inheritance re. user-defined classes

A class name in a class statement is followed by ( … ) when the parentheses content identifies inheritance classes = > in h4, each class is introduced with: class className:. object is the default parent class for a typical user-defined class, but (object) is usually omitted from: class className(object): to just class className:

\_\_init\_\_ parameters

A simpy class \_\_init\_\_ function signature usually begins with the 2 parameters self,env ;

if \_\_init\_\_ has additional parameters, they follow self and env.

A pf in a class has a signature starting parameter self. Other parameters, if any, will follow self.

The code self.x anywhere in a class references item x of a class instance.

Class definitions and class instantiations

classInstanceName = className(arguments) creates a

class instance named classInstanceName. The \_\_init\_\_

is automatically executed, and initializes an instance of (some or all of) the class’s variables, etc.

The class instantiation argument list does NOT include self, but must include env to access simpy properties.

Beginning part of a warehouse class definition

class Warehouse:

""" Roofing materials Warehouse operations """

def \_\_init\_\_(self,env,warehouseName):

self.env = env

self.warehouseName = warehouseName

pdb.set\_trace()

self.numberOfHD = 3 # Number of HD stores to be supplied

: *< -- The “:” is instructor notation for the rest of the class’s code*

An instance of class Warehouse is created by a statement such as:

w = Warehouse(env,'WH1') # Create the warehouse instance named "w"

Debugging with default Python debugger pdb

The example trace of idle output display below indicates arrival at the breakpoint, and simple examination of some class instance values.

=========== RESTART: C:\Users\bill\148\_s19\homework\h4\_148\_f19.py ===========

> c:\users\bill\148\_s19\homework\h4\_148\_f19.py(22)\_\_init\_\_()

**:** *< -- Some lines not displayed*

(Pdb) l *< -- “l” abbreviates “list some source lines around this breakpoint*

17 def \_\_init\_\_(self,env,warehouseName):

18 self.env = env

19 self.warehouseName = warehouseName

20 self.numberOfHD = 3 # Number of HD stores to be supplied

21 pdb.set\_trace() # When any set\_trace() call is reached, a breakpoint occurs

22 : *< -- Some lines not displayed*

(Pdb) numberOfHD *< --* Typing an item displays its value

\*\*\* NameError: name 'numberOfHD' is not defined

(Pdb) self.numberOfHD

3

(Pdb) self *< -- This class instance’s internal id*

<\_\_main\_\_.Warehouse object at 0x000002DE8828DE80>

(Pdb) self.\_\_dict\_\_ < -- \_\_dict\_\_ displays the current “properties” of an object

{'env': <simpy.core.Environment object at 0x000002DE81A2E6D8>, 'warehouseName': 'WH1', 'numberOfHD': 3}

(Pdb) quit *< -- Exit pdb AND abort script execution*

*Some very verbose output is posted – Python is simply displaying (by default) a stack unwind trace;*

*You will always get such a trace for execution time errors, as well.*

Traceback (most recent call last):

File "C:\Users\bill\148\_s19\homework\h4\_148\_f19.py", line 52, in <module>

w = Warehouse(env,'WH1') # Create the warehouse instance

File "C:\Users\bill\148\_s19\homework\h4\_148\_f19.py", line 22, in \_\_init\_\_

self.initHD\_inventory = 200 # 200 shingles bags at start of each HDk operations

File "C:\Users\bill\148\_s19\homework\h4\_148\_f19.py", line 22, in \_\_init\_\_

self.initHD\_inventory = 200 # 200 shingles bags at start of each HDk operations

bdb.BdbQuit

New, 11/14 pm) - **Using a list for scheduling multiple pfs**

There are many awkward ways (and a couple of good ways) to code situations where many (5, 400, etc.) pfs must be scheduled by simpy. In h4 each of the HD class instances must schedule execution of their inventory processing pf.

*A major hint about the HD class is: HD should define a pf that models the drawdown/decrement of an HD’s shingles bags inventory. Such a pf simply decrements current inventory level I(t) every 1 or 2 hours by a random order size in ~~[6,35]~~ [10,45], and reports the revised/updated I(t) level.*

The rest of this section is a simple simpy code example that illustrates a good way to schedule (any number of) pfs. It avoids awkward approaches that would create a distinct Python variable for naming each pf to be scheduled. (If we wanted 35 HDs, it would be tedious to have 35 different variable names, one for each pf instance to be scheduled.

= => Adapt any ideas from the following demo that helps in coding h4.

SOURCE CODE

"""

Demo on scheduling multiple simpy pfs - multiPF\_sched.py

WJM, CSUS CSC148, fall 2019

"""

import simpy

env=simpy.Environment() # simpy environemnt class instance

def x(env,p):

“”” Do something for p\*p time units “”” *< -- On athena, double quotes should NOT be slanted as happens in Windows*

print("x's parameter p has value ",p)

yield env.timeout(p\*p) # Do something for p\*p t.u.

print("x occurrence with parameter p = ",p," finished at time ",env.now)

pfList = [] # Initialize a list of scheduled x executions

# Schedule 5 executions of pf x

for k in range(5):

pfList.append(env.process(x(env,k))) # x(env,k) execution is scheduled, and the pf call is stored in pfList

print("\n pfList contains ",pfList) # See the list's pf objects

env.run() # Run until no more events are scheduled

MODEL RUN RESULTS

=========== RESTART: C:/Users/bill/148\_f19/simpy3/multiPF\_sched.py ===========

pfList contains [<Process(x) object at 0x2e51ddde898>, <Process(x) object at 0x2e51de5a978>, <Process(x) object at 0x2e51de67160>, <Process(x) object at 0x2e51de90470>, <Process(x) object at 0x2e51de908d0>]

x's parameter p has value 0

x's parameter p has value 1

x's parameter p has value 2

x's parameter p has value 3

x's parameter p has value 4

x occurrence with parameter p = 0 finished at time 0

x occurrence with parameter p = 1 finished at time 1

x occurrence with parameter p = 2 finished at time 4

x occurrence with parameter p = 3 finished at time 9

x occurrence with parameter p = 4 finished at time 16

>>>

You can use the pfList.append() call in the above source code as a first of two steps to Schedule HD’s pfs.

pfList.append(env.process(x(env,k))) directly schedules execution of an x pf instance each time through the loop (a total of 5 times). h4 needs to schedule a pf of each HD instance.

First Step: 3 instances of the HD class are created first (before their pf occurrences can be created). Each HD instance that is created has a different argument for that HD’s name, so each HD’s name must be an argument passed to HD’s \_\_init\_\_() function. This means that pfList.append() can have as its argument the instantiation of an HD instance (rather than an env.process() call as in the above code).

Thus, the jth pfList.append() call will store a reference to the jth HD instantiation into the jth pfList location. Recall from class discussions/examples that the append modifies (adds an element to) the list, and so it suffices to call pfList’s append function. That is, assignment to a variable as in: z = pfList.append(xxx) is irrelevant, since the “z = “ part of this statement is not needed in this situation.

Second Step: With the 3 HD class instances now existing, simply add a second line of code to schedule each HD’s pf (assume you named the pf doOrders) with: env.process(pfList[j].doOrders(env)).

{ the approach just described works for scheduling 1 or 3 or 1000 pfs, eliminating the need for a local variable reference to each pf instance; the list’s elements become the pf references }