**Process Modeling in simpy CSC148,** Nov 7, 2019

Review python function basics

Using the def keyword, specify the name of a function, with parameters in parentheses.

(Parameterless functions require an empty pair of parentheses.)

>>> def foo(x): *< -- Colon terminates fcn header, and starts the scope of the function body*

*… “”” foo prints contents of one passed argument*

*… “””*

... print (x) *<- - Body of fcn is a block that must be indented, just like other source code blocks*

...

>>> foo(123) *<- - A call to fcn foo*

123

A function call specifies a function name and a pair of parentheses, putting any requirement arguments inside.

Moreover, in the above print statement, if x was a list, instead of a scalar, the list contents would be printed.

As we see next, a function is a “first class” object.

>>> foo

<function foo at 0x0000012175226D90>

As such, an identifier for a function can be passed as an argument for some other function, or stored in a list, etc.

simpy simulations

This module discusses and demos execution of a first DES model code in simpy3 (abbreviate simpy).

gpssW experience makes the simpy “learning curve” much shorter because of knowledge already gained from the first half of this course about DES systems.

And, as in most DESs, the user comments specify the “time unit (aka t.u.)” in each simulation.

*(Note: it is possible to implement simulations in simpy where the model t.u. is related to real-world wall-clock time, using the asyncio module. We will not do this form of simulation in CSC148)*

Def - a process function is a python function that references a simpy environment with syntax such as: simpyEnvironmentReference.someEnvironmentAttribute. Such a function simulates some activity in a simulation.

gpssW users will recognize the rough equivalence of a gpssW “transaction” (code: GENERATE … TERMINATE) to a simpy “process function”.

#1 Some (as close as possible) equivalent constructs in simpy and gpssW

Define a required reference for a simulation environment to be able to access simpy capabilities: Example:

env = simpy.Environment() No direct equiv. in gpssW, other than part of the effect of Command - -> Create Simulation

In the above name binding, “env” can be any identifier; it is not a python/simpy keyword. Almost all simpy documentation consistently uses the variable env as a simpy Environment handle, thus = >

* CSC148 will follow the convention that “the” simpy environment (handle) name is “env”

Schedule a “process function” for execution by the DES scheduler:

env.process(doNothing()) Similar to part of effect of gpssW GENERATE except that GENERATE automatically schedules occurrences of the process it represents.

(if a scheduled function, such as doNothing has parameters, they must be specified in env.process call)

The body of function doNothing defines whether doNothing does one set of activities one time, or repeats an activity a specified number of times (including no limit when the code contains the loop syntax while True:)

That is, in simpy, there is more fine-grained control (vs. a gpss tr) over how/when a scheduled function executes.

The phrase “process function” occurs frequently with simpy , so abbreviate is by “pf”.

Model some simulated time to do some activity

yield env.timeout(numeric expression) *Note! Env.timeout(x) alone only creates a time-based object*

(Like gpssW ADVANCE numeric expression, but yield has many variations not shown here)

(Retrieving) current model time

env.now (not env.time – this is illegal/undefined) returns current model time gpssW equivalent: AC1

Model simulation start and run duration

env.run( ) gpssW equivalent: Start n and tc initialization

run is “callable”; that is, run() is a function call, so “()” are required; there are several parameter options:

If no argument is given, as above, the simulation halts whenever there are no scheduled future events.

Argument is until=n has an unusual effect: the simulation runs until all events scheduled BEFORE model time n

Example: env.run(until=20)

Argument is until= a pf name (or the name of a pf instance); until=pf (or a named pf occurrence) means terminate the simulation after a pf or the named pf occurrence has executed.

Example: env.run(until=somePF\_instance)

#2 Overall structure of a simpy model

A process function can be considered as an “event”, but in a very general way. simpy events can be either instantaneous or have long duration.

===================================================================================

It is important to note that each pf must contain a yield statement of some kind (there are several variations of the yield statement) so it can be processed by the simpy event scheduler;

====================================================================================================

Process function calls will pass the argument env and its def signature contains env as well.

In many simulations, such env occurrences are not needed, but for consistency, CSC148 follows this env use.

One or concurrent (>=2) simulation runs in 1 script are possible; however, concurrent runs need distinct simpy Environment handles. We will execute one simulation at a time.

#3 Summary of source code layout of a basic simpy model

1. import simpy
2. define process functions, and any other “ordinary” python code needed for a simulation
3. define env (can appear anywhere in the code) (not required to appear first in the code)), but must precede statements that schedule pfs
4. initialize all variables, as in any language
5. using env.process() calls, schedule all relevant process functions *(env.process() calls can occur anywhere in code, at places that will correctly order pf actions)*
6. env.run() along with any parameters appearing inside the () … launches a simulation & the run parameter specifies the run termination condition.

When developing any new model, and just like with ordinary programming, diagnostic print() statements facilitate event tracing.

Along with crude print debugging, there is a simple, but useful built-in debugger for python+simpy named pdb. To use, pdb, place: import pdb at top of a module . Any place in the code where the call pdb.set\_trace() in encountered at execution time, execution will be suspended, and dbg commands can be used.

#4 Outline of introductory demos

*The first script illustrates defining/using generators – it is NOT a simpy simulation*

Ordinary python functions, generators, and generator occurrences are demoed. Note the required use of yield in the generator named gen.

= = > Discuss FIRST DEMO below

*The second program is our first genuine simpy simulation (that uses: import simpy)*

= = > Discuss SECOND demo below

*The third program is a slightly more complex simpy simulation (2 pfs are scheduled)*

= = > Discuss THIRD DEMO below Source code filename: processFunction\_demo.py

Each env.process call schedules its argument function (that must be a pf)

for execution/startup according to additional optional arguments.

The default is that the process function is scheduled for execution at model time env.now.

*Note: as soon as env is defined in a script execution, process functions can be scheduled for execution using env.process() calls.*

\* abc() is illegal, if it is intended to be a pf

\* A pf contains arbitrary modeling code for some activities/actions, much like

the blocks in a gpssW transaction.

\* pf doNothing is a trivial stub pf that shows the bare minimum creation of a schedulable function.

\* pf example has more substance, in that, when execution starts (and this is concurrently at model time 0 with

doNothing) , example starts executing a loop. The loop illustrates how to implement a repetitive activity.

\* ignoring the value = left side part of the yield statement below,

and assuming a pf is intended to repeatedly do a particular activity, the statement

value = yield env.timeout(arguments)

implements an amount of time to simulate ONE repetition of some simulated activity;

the yield would appear in the body of a loop in the pf’s code

*Recall the behavior of yield:: a pf first executes to a first encountered yield, returns a value to the caller, but then, the pf does not terminate execution as does an ordinary function in most other languages; all pf context still exists; when this pf is again called, it resumes execution right after the previous yield and executes until a yield s again encountered*

*(termination of a pf occurs when the yield can no longer execute)*

**FIRST DEMO**

Example of building generator occurrences

"""

Compare "ordinary" functions with generators - superSimple\_generator.py (*A copy is in Athena directory:* /gaia/home/faculty/mitchell/simpy3demos)

Note: simpy pfs are based on properties of generators

"""

# An ordinary python function

def fun():

return -17 *< -- More generally, return x,y, … returns several values in a tuple (not a list)*

# A python generator

def gen():

“”” Each next call to a gen instance returns yield argument k, and suspends;

each subsequent next call to the occurrence resumes execution until yield is encountered again “””

for k in range(5):

yield k # The kth next call to an instance of gen returns int value k

print("”” The next call after the loop with the yield sends back to caller

the value of the generator's return statement AND throws a StopIteration error”””)

return 47

print("Value returned by call to ordinary function fun: ",fun())

print("Show gen is a generator, not an ordinary function by 'calling it' in usual way ")

print(" 2 gen 'Calls' create 2 anonymous instances of gen: ", gen(), gen())

# Create 2 named generator instances based on generator gen

gen\_iter1 = gen()

gen\_iter2 = gen()

print("Display the generator instance locations")

print( "gen\_iter1 ", gen\_iter1, " gen\_iter2 ", gen\_iter2 )

print("Each NAMED gen instance can use next(genInstanceName) to get the next yield argument value")

print(" Apply enough next calls to generator gen\_iter1 to exhaust the yield loop ")

for i in range(6):

print(next(gen\_iter1))

Execution Results –

======= RESTART: C:\Users\bill\148\_f19\simpy3\superSimple\_generator.py =======

Value returned by call to ordinary function fun: -17

Show gen is a generator, not an ordinary function by 'calling it' in usual way

2 gen 'Calls' create 2 anonymous instances of gen: <generator object gen at 0x0000029F6FA2C620> <generator object gen at 0x0000029F6FA2C5C8>

Display the generator instance locations

gen\_iter1 <generator object gen at 0x0000029F6FA2C620> gen\_iter2 <generator object gen at 0x0000029F6FA2C5C8>

Each NAMED gen instance can use next(genInstanceName) to get the next yield argument value

Apply enough next calls to generator gen\_iter1 to exhaust the yield loop

0

1

2

3

4

The next call after the loop with the yield sends back to caller

the value of the generator's return statement AND throws a StopIteration error

Traceback (most recent call last):

File "C:\Users\bill\148\_f19\simpy3\superSimple\_generator.py", line 35, in <module>

print(next(gen\_iter1))

StopIteration: 47

>>>

**SECOND Demo**

"""

demoSimplePF.py pfs Note: Usually a pf does NOT need (nor, as with Python) require RETURN

"""

import simpy # simpy runtime

import pdb # Default Python debugger

env=simpy.Environment()

def x(env):

""" x repeats its loop body 3 times """

for k in range(1,4):

print("Starting x loop body execution ",k," at time ", env.now)

yield env.timeout(2)

print("Finished an x YIELD at time ", env.now)

print("pf x execution finished at time ", env.now,"\n")

def y(p):

print("y executing with arg", p,"\n")

y(3) # An ordinary fcn can be called

#x(env) # NOTE! merely calling x does nothing except post the end of simulation msg.

env.process(x(env)) # Schedule x at time 0

env.run()

print("No more scheduled events ... simple PF demo ends")

Execution Results –

=========== RESTART: C:\Users\bill\148\_f19\simpy3\demoSimplePF.py ===========

y executing with arg 3

Starting x loop body execution 1 at time 0

Finished an x YIELD at time 2

Starting x loop body execution 2 at time 2

Finished an x YIELD at time 4

Starting x loop body execution 3 at time 4

Finished an x YIELD at time 6

pf x execution finished at time 6

No more scheduled events ... simple PF demo ends

>>>

NOTE!!!

If env.process(x()) is replaced with: x() function x() is NOT scheduled:

Execution Results –

=========== RESTART: C:\Users\bill\148\_f18\simpy3\demoSimplePF.py ===========

No more scheduled events ... demoSimplePF ends

**THIRD DEMO**

"""

Source code processFunction\_demo.py

Refr http://simpy.readthedocs.io/en/latest/topical\_guides/simpy\_basics.html

WJM modification of the first simpy 'ReadTheDocs' documentation

Illustration of multiple process functions that model schedulable actions/activities

Each simpy entity that is an event, the state an event is ONE of:

untriggered: means: not known to the event queue

triggered: means: scheduled at a time t and inserted into the event queue

processed: means: removed from the event queue

"""

import simpy

def abc(env):

"""

This is an ILLEGAL pf; it is not enough to pass env as a parameter, and

reference a property of env; each pf must contain >=1 yield statement, in order

for it to be, first and foremost, a generator object

To see the runtime error, uncomment the call: env.process(abc(env)), and run the model

"""

print("Function abc was executed at time ", env.now)

def doNothing(env):

"""

Dummy pf

"""

yield env.timeout(0) # It takes 0 time to do nothing

print("fcn doNothing did nothing, and finished at time ", env.now)

def example(env,p\_delay, p\_eventResultValue):

"""

Demo a simpy event e in the form of a function; to be schedulable, e MUST contain >=1 yield

The loop illustrates that a generator-iterator (aka generator) executes its code to

the next encountered yield (either via successive fcn calls or >1 yield in its own body).

Each yield returns a value to caller.

Also, by default Python parameters are passed using "object result" mode.

See output item 4.

"""

for k in range(1,3):

print("Starting fcn example at time ", env.now)

value = yield env.timeout(p\_delay,value=str(p\_eventResultValue)+' from Iter# '+str(k)) # Return yield value

print("Value returned by fcn example yield is ", value, " after delay ", p\_delay, "at time ", env.now)

p\_eventResultValue += 1000 # Update the fcn's formal parameter

print("env's event queue after example yield#", k, "\n", env.\_queue)

env = simpy.Environment() # Define simulation environment handle

doNothing\_id, example\_id = id(doNothing), id(example)

print(" <<< . . . Think of simpy script execution as happening top-down ... >>> \n")

print("doNothing & example are genuine objects w unique (during lifetime) ids: ",doNothing\_id,example\_id)

print("0. Configuring simulation structure and define 2 untriggered events")

print("1. Schedule but do not yet 'trigger' the process event named doNothing, at time ", env.now)

print("2. Schedule but do not yet 'trigger' the process event named example, at time ", env.now)

#env.process(abc(env)) # Illustrates an illegal pf definition

env.process(doNothing(env))

example\_arg = -1775 # Create an argument to pass to pf example

env.process(example(env,2,example\_arg))

print("env's event queue after doNothing and example are created/untriggered:")

for x in env.\_queue:

print(x)

# run, with no until parameter, executes until there are no more events scheduled

# until=x parameter runs until time t, but will NOT process scheduled events at time x,

# so specify an until= value that is a little larger than x

print("3. SIMULATION STARTING at time ", env.now)

env.run(until=5)# Launch the simulation

print("4. After example() finished, example\_arg has retained its value from before pf execution: ", example\_arg)

print("env's event queue after doNothing() and example() were processed \n", env.\_queue)

print("5. SIMULATION ENDED at time ", env.now)

The fatal runtime error that occurs if abc() is called

Traceback (most recent call last):

File "C:\Users\bill\148\_f18\simpy3\processFunction\_demo.py", line 45, in <module>

env.process(abc(env))

File "C:\Users\bill\AppData\Local\Programs\Python\Python36\lib\site-packages\simpy\events.py", line 311, in \_\_init\_\_

raise ValueError('%s is not a generator.' % generator)

ValueError: None is not a generator.

======================================================================================================

Execution results when abc() is not called:

<<< . . . Think of simpy script execution as happening top-down ... >>>

doNothing & example are genuine objects w unique (during lifetime) ids: 2427881029008 2427881280240

0. Configuring simulation structure and define 2 untriggered events

1. Schedule but do not yet 'trigger' the process event named doNothing, at time 0

2. Schedule but do not yet 'trigger' the process event named example, at time 0

env's event queue after doNothing and example are created/untriggered:

(0, 0, 0, <Initialize() object at 0x23548f4e748>) *<-- Schedule doNothing() and example() events at time 0*

(0, 0, 1, <Initialize() object at 0x23548fc1e48>)

3. SIMULATION STARTING at time 0

Starting fcn example at time 0

fcn doNothing did nothing, and finished at time 0

Value returned by fcn example yield is -1775 from Iter# 1 after delay 2 at time 2

env's event queue after example yield# 1

[(5.0, 0, 2, <Event() object at 0x23548f3ea58>)]

Starting fcn example at time 2

Value returned by fcn example yield is -775 from Iter# 2 after delay 2 at time 4

env's event queue after example yield# 2

[(5.0, 0, 2, <Event() object at 0x23548f3ea58>)] *<-- Run termination is scheduled at time 5*

4. After example() finished, example\_arg has retained its value from before pf execution: -1775

env's event queue after doNothing() and example() were processed

[]

5. SIMULATION ENDED at time 5.0

>>>

pf synchronization

Advanced simpy simulations usually need to implement various ways that different pfs interact.

As in everyday activities:

1. pf#1 needs to be scheduled to happen/occur

To be able to refer to a scheduled fp instance, must do: pfInstanceName = env.process(pfName(arguments))

Then, anywhere in the model code, the instance (while it exists) can be referenced.

If such a reference is not needed, the assignment is not needed.

1. pf#1 must wait for another pf#2 to either do something or finish

Always done using a yield statement

The most common wait situations are:

1. pf#1 waits for pf#2 to finish and b) pf#1 waits for a shared event to occur (See item 3.)
2. pf#1 creates/defines an event or state that can be coordinated with other pfs (*simpy calls these “shared events”*)

It should be possible for any number of fps (including the shared event creator) to wait for a shared event to occur

Create a named shared event by: seName = env.event()

a shared event is GLOBAL in simpy

1. pf#1 must resume after pf#1 finishes waiting for something to happen, that is, the waited-for event has

occurred; resuming will occur automatically/instantaneously after the event happens

1. pf#1 interrupts pf#2

After pf#1 interrupts pf#2, pf#1 asynchronously continues execution

As with hardware interrupts (and most human interruptions), the simpy code to handle an interrupt should be as efficient and fast as possible. Interrupt-handling code/activity generally has priority over normal (non-interrupted) code execution

Concerning 5. above, an interrupt, by itself, is not useful. There must be a way for the interrupted #2 to determine the cause of the interrupt, and then do appropriate actions in its interrupt handler.

simpySynch\_demo.py

This execution demo illustrates the scenarios outlined in this document’s section titled pf synchronization.