SimPy

Sistema para simulación de eventos discretos

SimPy

- Es un sistema para la simulación de eventos discretos
- Está escrito en Python.
- Ofrece diversas herramientas para escribir programas de simulación.
  - Procesos
  - Recursos
  - Monitores

Proceso

- Son las entidades activas de la simulación
- Heredan de la clase *Process* de SimPy.
- Pueden:
  - Solicitar recursos.
  - Hacer colas esperando por un recurso
  - Detener su operación por tiempos fijos o aleatorios
  - Puede ser interrumpido por o interactuar con otros Procesos.

Procesos

- Para crear Procesos en SimPy
  - *class nombreProceso*(Process):
    - La clase *Process* tiene un atributo name.
  - Debe definirse en el proceso al menos un Método de Ejecución del Proceso (PEM por sus siglas en inglés)
  - El PEM define las acciones a ser llevadas a cabo por el proceso.
  - Debe contener al menos una instrucción *yield*.
    - Afecta el curso del ciclo de vida de el proceso.
    - Controla la ejecución y sincronización de múltiples Procesos.

Procesos (instrucción *yield*)

- Instrucción de SimPy que permite:
  - Congelar la ejecución del PEM de el proceso, hasta que se cumpla una condición.
    - Una vez cumplida se continúa la ejecución en la siguiente instrucción del PEM.
  - Controlar la ejecución y sincronización de múltiples Procesos.

Definiendo un proceso

```
class Carro(Process): #entidad carro que hereda de la clase Process
    def __init__(self, nombre, cc): #método constructor
        Process.__init__(self, name=nombre)
        self.cc=cc #nuevo atributo capacidad del motor
    def go(self): #PEM
        print now( ), self.nombre, “Comenzando”
        yield hold, self, 100.0 # se congela por 100 unidades de tiempo
        print now( ), self.nombre, “Llegando”
```

Instanciar Procesos

Se crean objetos de la clase carro creada anteriormente

```
c1 = Carro(“El rayo”, 2000) #un nuevo carro
c2 = Carro(“Mac”, 1600) # otro nuevo carro
```

Activar Procesos

- Activar un proceso implica planificar eventos para este en un tiempo t de la simulación
- Para ello se usan la instrucción *activate* o el método *start* de la clase que representa el proceso.

```
activate(c1, c1.go( ), at=6.0) # activa c1 en el tiempo 6.0
c2.start(c2.go( ))       # activa c2 en el tiempo 0.0
```

Activar Procesos

Sintáxis

Activate (*p, p.pemnombre*([*arg*]), {at=*t*|delay=*periodo*}
[,prior=*prior*]

p.start (*p.pemnombre*([*arg*]), {at=*t*|delay=*periodo*}
[,prior=*prior*]

- **at**= el tiempo en el que el proceso sera activada, por omisión es el tiempo actual (at=now ( ))
- **delay**= Periodo de tiempo en que se demora la activación
  por omisión 0.0
- **prior** = prioridad de el proceso (se usa en las colas), puede
  ser True o False, este último es el valor por omisión.

Procesos

- Tambien se pueden usar comandos para poner en reposo (passivate), reactivar o cancelar (sacar del planificador) para controlar las Procesos
  - yield passivate,self
  - reactivate(p [, {at=*t*|delay=*periodo*}] [,prior=*prior*] ?)
  - self.cancel(p)

Modelación de una Simulación

- Programa realizado en Python
- Debe importar el módulo de simulación
  - from SimPy.Simulation import *
  - from SimPy.SimulationTrace import *
- Obtener el tiempo actual en la simulación
  - Funcion now( )

Modelación de una Simulación

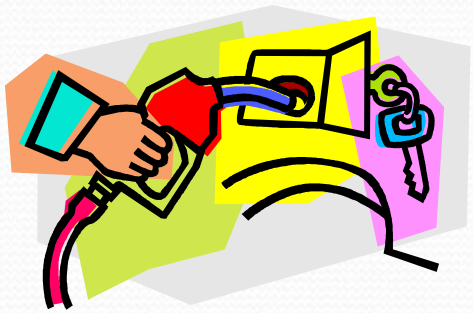
- Iniciar el sistema
  - inicialize ( )
- Comenzar la simulación
  - simulate(until=*tiempo_fin*)
    - donde tiempo_fin es el tiempo que durará la simulación
- Terminar la simulación
  - stopSimulation( )

Ejemplo de un Modelo de Simulación

```
from SimPy.Simulation import *

class Carro(Process):
    def  __init__(self, nombre, cc): #este es el método constructor
            Process.__init__(self, name= nombre)
            self.cc=cc
    def  go(self): #este es el PEM
            print now( ), self. nombre, “Comenzando”
            yield hold, self, 100.0
            print now( ), self. nombre, “Llegando”

initialize ( ) #inicializa las variables globales y el tiempo.
c1 = Carro(“El rayo”, 2000) #un nuevo carro
c2 = Carro(“Mac”, 1600) # otro nuevo carro

activate(c1, c1.go( ), at=6.0 # activa c1 en el tiempo 6.0
c2.start(c2.go( ))      # activa c2 en el tiempo 0.0
simulate(until=200) #comienza la simulación hasta el tiempo 200 o hasta que no haya
    más eventos planificados
print ‘El tiempo actual es ‘, now( )
```

Recursos

- Son los componentes pasivos de la simulación.
- Constituyen puntos de congestión (encolamiento) de capacidad limitada.
- SimPy tiene 3 tipos(clases):
  - Resource.
  - Level
  - Store
- Los Procesos deben solicitar un recurso y el recurso los encola si no está disponible



Recursos

- Para definir un recurso

  r= Resource (capacity=n, name='nombre', unitName='unidad', qType=*Tipo*, preemtanble=False|True, monitores= False|True, monitorType=Monitor)

  Donde

  - capacity número real o entero positivo que especifica el número de unidades idénticas dentro del recurso
  - name Nombre descriptivo del recurso (p.e. 'banco')
  - unitName Nombre descriptivo para las unidades de recurso (p.e 'taquillas')
  - qType Diciplina de espera en la cola del recurso, puede ser FIFO o PriorityQ, si no se especifica toma el valor FIFO.
  - preemtable Si es verdad, un proceso con mayor prioridad puede sacar del recurso a una de menor prioridad.
  - monitored Si es True, se recoge información de estado sobre el tamaño de las colas

Recursos

- Automáticamente SimPy crea dos colas en el recurso
  - waitQ la cual es la cola de Procesos en espera por usar el recurso
  - activeQ la cual es la cola de Procesos que actualmente usan una unidad del recurso (el largo máximo de esta es la capacidad del recurso)

Recursos
Información

- r.n da el número de unidades libres
- r.waitQ una cola (lista) de Procesos en la cola de espera, len(r.waitQ) tamaño de la cola
- r.activeQ una cola (lista) de Procesos usando una de las unidades del recurso, len(r.activeQ)
- r.waitMon registro (hecho por un monitor) de la actividad en r.waitQ
  - r.waitMon.timeaverage( )
- r.activeMon registro (hecho por un monitor) de la actividad en r.activeQ

Recursos
Solicitud y liberación

- Una entidad puede solicitar y luego liberar una unidad de un(os) recurso(s) usando los siguientes comandos yield en su(s) PEM.
  - yield request, self,r[P=0] #r es el recurso

    P es opcional y define la prioridad, mayores valores de P definen mayor prioridad.
  - yield release, self, r

Recursos
Orden en la cola

- Una entidad solicitante debe esperar en la cola en un orden determinado por los atributos *qType* y *preemtable* del recurso y el valor de *P* en la solicitud
  - Si qType es FIFO la primera que llega será la primera en ser atendida y las prioridades serán ignoradas, más aun en este tipo de cola no hay preferencias.
  - Si qType es PriorityQ las prioridades en las solicitudes son reconocidas.

Registros de las colas Recursos

- Se pueden llevar registros de las colas asociadas a cada recurso. SimPy usa para esto los Monitores.
- Si un recurso r es definido con *monitored=True*, SimPy registra automáticamente la longitud de sus colas waitQ y activeQ y son almacenados en los objetos de registro llamados r.waitMon y r.activeMon

Registros de las colas Recursos

```
print ‘Promedio de espera pesado en tiempo:’
      ,r.waitMon.timeAverage()
print ‘Promedio de recursos en espera:’
      ,r.waitMon.mean()
print ‘varianza de espera:’ ,r.waitMon.var()
print ‘Tiempo promedio de servicio:’
      ,r.activeMon.timeAverage()
print ‘Promedio de resursos servidos:’
      ,r.activeMon.mean()
print ‘SD de recursos servidos:’
      ,sqrt(r.activeMon.var())
```

Monitores

- Existen dos tipos de monitores:
  - Monitor
  - Tally
- Observan y almacenan el valor de una variable de interés y retornan datos estadísticos durante, o al terminar la simulación.
- Usan el método *observe* para guardar los valores.
- Se pueden usar p.e para almacenar tiempos de espera para una secuencia de clientes en una tienda

Monitores

- Un nuevo monitor

  miMonitor=Monitor(name=‘*nombre*’ ,
  ylab=‘*y*’, tlab=‘t’)

  - *name:* es un nombre descriptivo para la instancia de Monitor, por omisión es ‘a_Monitor’
  - *ylab y tlab:* son etiquetas descriptivas usadas por el paquete SimPlot para dibujar datos.

Monitores

- Registrando datos

  Para observar una variable en un instante *t* y guardar su valor se usa el método *observe.*

  - miMonitor.observe(v, [,t])
  - Guarda el valor actual de la variable *v* y el tiempo *t* (o el tiempo actual *now( )* si se omite*).*
  - Para estar seguro de que el promedio en tiempo sea calculado correctamente se debe llamar inmediatamente luego de un cambio en la variable.

Resúmenes de datos

Comando	Descripcion
count( )	Numero actual de observaciones
total( )	Suma de los valores observados
mean( )	Promedio de valores almacenados sin tomar en cuenta el tiempo en fueron guardados (m.total()/m.count())
timeAverage( )	Promedio de valores guardados pesados en el tiempo
var	Varianza simple de Procesos en la cola (observaciones)
timeVariance	Varianza de tiempo en cola
reset( )	Reinicia el monitor
str( )	Texto que describe el estado actual del monitor, se puede usar en una instrucción print

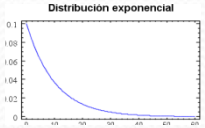
Generación de números aleatorios

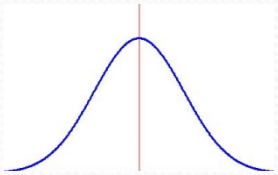
- SimPy no suministra generadores de números aleatorios.
- Se requiere importar el módulo *random* de Python.
  - from random import seed, random, expovariate, normalvariate
- Seed representa la semilla, la cual conviene, en algunos casos, fijar en un valor inicial.

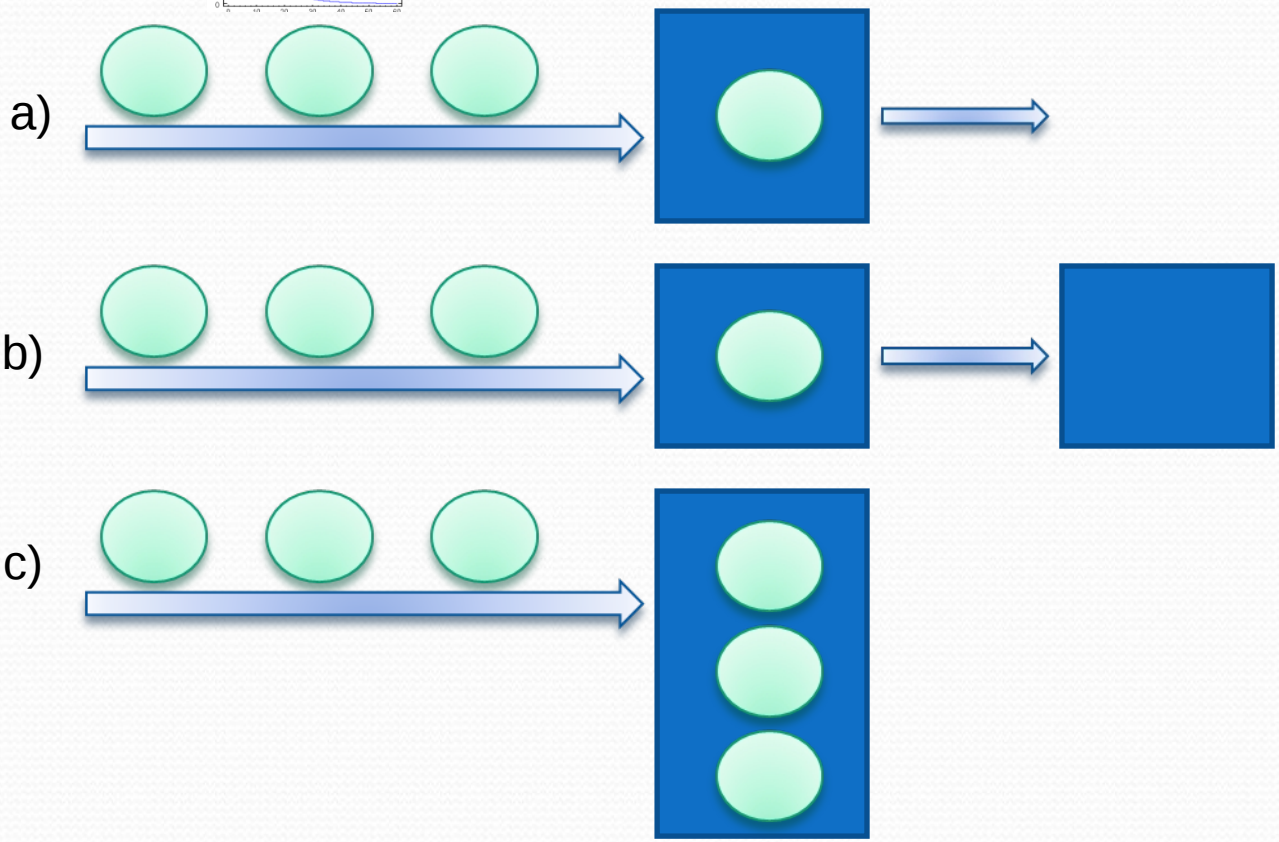
Generación de números aleatorios

```
from ramdom import seed, expovariate,
   normalvariate

seed(333555)

X=expovariate(0.1)

Y=normalvariate(10.0, 1.0)
```

Modelos







Modelos (ejemplos)

```
from SimPy.Simulation  import *

class Cliente(Process):
        """Simula un cliente de un cajero"""
        def __init__(self, nombre):
                super(Cliente, self).__init__(nombre)

        def realizar_transaccion(self,tiempo_transaccion=30.0):
                yield request, self, r1
                print self.name, "Comenzo la transaccion en ", now()
                yield hold, self, tiempo_transaccion
                yield release, self, r1
                print self.name, "Termino la transaccion ", now()

tiempo_maximo = 500.0
r1 = Resource(name="Cajero")

initialize()

c1=Cliente(nombre="Alfons")
activate(c1,c1.realizar_transaccion(tiempo_transaccion=35.0),at=5.0)
c2=Cliente(nombre="Kaspar")
activate(c2,c2.realizar_transaccion(tiempo_transaccion=17.0),at=2.0)
c3=Cliente(nombre="Giuseppe")
activate(c3,c3.realizar_transaccion(tiempo_transaccion=20.0),at=7.0)

simulate(until=tiempo_maximo)
```

Modelos (ejemplos)

```
from SimPy.Simulation  import *

class Cliente(Process):
        """Simula un cliente de un cajero"""
        def __init__(self, nombre):
                super(Cliente, self).__init__(nombre)

        def realizar_transaccion(self,tiempo_transaccion=30.0, tiempo_compra=30.0):
                yield request, self, r1
                print self.name, "Comenzo la transaccion en ", now()
                yield hold, self, tiempo_transaccion
                yield release, self, r1
                print self.name, "Termino la transaccion en cajero en ", now()
                yield request, self, r2
                print self.name, "Comenzo la compra en cafetin en ", now()
                yield hold, self, tiempo_compra
                yield release, self, r2
                print self.name, "Termino la compra en cafetin en ", now()

tiempo_maximo = 500.0
r1 = Resource(name="Cajero")
r2 = Resource(name="Cafetin")
initialize()
c1=Cliente(nombre="Alfons")
activate(c1,c1.realizar_transaccion(tiempo_transaccion=35.0, tiempo_compra=30.0),at=5.0)
c2=Cliente(nombre="Kaspar")
activate(c2,c2.realizar_transaccion(tiempo_transaccion=17.0, tiempo_compra=30.0),at=2.0)
c3=Cliente(nombre="Giuseppe")
activate(c3,c3.realizar_transaccion(tiempo_transaccion=20.0, tiempo_compra=30.0),at=7.0)
simulate(until=tiempo_maximo)
```

Modelos (ejemplos)

```
from SimPy.Simulation  import *

class Cliente(Process):
        """Simula un cliente de un cajero"""
        def __init__(self, nombre):
                super(Cliente, self).__init__(nombre)

        def realizar_transaccion(self,tiempo_transaccion=30.0):
                yield request, self, r1
                print self.name, "Comenzo la transaccion en ", now()
                yield hold, self, tiempo_transaccion
                yield release, self, r1
                print self.name, "Termino la transaccion ", now()

tiempo_maximo = 500.0
r1 = Resource(capacity=3, name="Cajero")

initialize()

c1=Cliente(nombre="Alfons")
activate(c1,c1.realizar_transaccion(tiempo_transaccion=35.0),at=5.0)
c2=Cliente(nombre="Kaspar")
activate(c2,c2.realizar_transaccion(tiempo_transaccion=17.0),at=2.0)
c3=Cliente(nombre="Giuseppe")
activate(c3,c3.realizar_transaccion(tiempo_transaccion=20.0),at=7.0)


simulate(until=tiempo_maximo)
```