Klasser vid simulering i Python

I föreläsningen om simulering presenterades en händelsecentrerad metod för att simulera ett kösystem. Det innebär att man har ett tillstånd som uppdateras när det inträffar händelser. Händelserna lagras i en lista (som kan implementeras som en heap). Se föreläsningen för en mer detaljerad förklaring.

Ofta är det bättre att använda ett mer objektorienterat synsätt där man definierar klasser som sedan kommunicerar med varandra genom att sända signaler. Man kan till exempel definiera en klass **generator** som genererar kunder, en klass **queue** som är ett kösystem och en klass **sink** dit man skickar alla kunder som har betjänats. Varje signal som skickas har en tidsstämpel som talar om när signalen ska komma fram. Signalerna lagras i t ex en heap så att det är enkelt att plocka fram den signal som har den minsta tidsstämpeln. Signalerna innehåller också en destination och ett tal som anger vilken typ av signal som det är.

I filen exempel.py finns det ett exempel på ett sådant program. Där finns en generator som genererar kunder, ett kösystem och en sink. Här nedan förklaras programmet del för del, först visas en del av programmet och därefter kommer en förklaring.

```
import heapq
import random
import numpy as np
```

Här importeras Pythons heap, slumptalspaket och numpy som är ett paket för numeriska beräkningar.

```
signalList = []

def send(signalType, evTime, destination, info):
    heapq.heappush(signalList, [evTime, signalType, destination, info])
```

En signallista, `signalList`, som är tom från början definieras. Därefter definieras en funktion send som använder Pythons egen heap för att skapa en signal och lägga in den i signalList. En signal är en lista `[evTime, signalType, destination, info]`.

- evTime = tiden när signalen ska komma fram
- signalType = vilken typ av signal det är, kan till exempel ange att en kund kommer eller är färdigbetjänad
- destination = vem som ska ha signalen
- info = annan information som man vill skicka med signalen

Signalerna sorteras med evTime som nyckel.

```
READY = 1
ARRIVAL = 2
MEASUREMENT = 3
simTime = 0.0
stopTime = 123456789.0
```

READY, ARRIVAL och MEASUREMENT är tre typer av signaler som används av programmet. simTime är en variabel som håller reda på tiden i programmet. Det är inte fråga om någon slags realtid utan

det är bara en variabel som uppdateras varje gång man tar ut en signal ur signallistan. stopTime anger när simuleringen ska sluta.

```
class larger():
    def __gt__(self, other):
        return False
```

Den här metoden behövs för att signallistan ska fungera. Den gör ”operator overloading” på relationen >. Utan den är det inte säkert att funktionen heap.heappush(signalList, signal) alltid kommer att fungera. De andra klasserna i programmet ärver denna klass.

```
class generator(larger):
    def __init__(self, sendTo):
        self.sendTo = sendTo
    def treatSignal(self, x, info):
        if x == READY:
            send(ARRIVAL, simTime, self.sendTo, simTime)
            send(READY, simTime + random.expovariate(1.0), self, None)
```

Denna klass genererar kunder. self.sendTo talar om vart kunderna som genereras ska skickas. Funktionen __init__(…) i en Python-klass initierar en instans av klassen. När en signal som ska behandlas av en instans av generator tas ut ur signalList så anropas funktionen treatSignal. Denna funktion måste finnas i alla klasser som ska kunna behandla signaler. I treatSignal så skickas först en ARRIVAL till self.sendTo. Tiden när kunden skickades iväg läggs till som info-parameter i send. Det är för att man ska kunna mäta hur lång tid det tar för kunden att komma fram till sink. Sedan skickar generatorn en signal READY till sig själv som kommer fram efter en exponentialfördelad tid. När den kommer fram genereras nästa kund osv. Observera att när jag skriver ”skickar en signal” så betyder det att signalen sorteras in i signalList, signalen behandlas inte av mottagaren innan den har plockats ut ur signalList.

```
class queue(larger):
    def __init__(self, sendTo):
        self.numberInQueue = 0
        self.measuredValues = []
        self.sendTo = sendTo;
        self.numberBlocked = 0
        self.numberServed = 0
    def serviceTime(self):
        return random.expovariate(1.0)
    def treatSignal(self, x, info):
        if x == ARRIVAL:
            if self.numberInQueue == 0:
                send(READY, simTime + self.serviceTime(), self, None)
                self.numberInQueue = self.numberInQueue + 1
                self.info = info
                self.numberServed = self.numberServed + 1
            else:
                self.numberBlocked = self.numberBlocked + 1
        elif x == READY:
            self.numberInQueue = self.numberInQueue - 1
            send(READY, simTime, self.sendTo, self.info)
        elif x == MEASUREMENT:
            self.measuredValues.append(self.numberInQueue)
            send(MEASUREMENT, simTime + random.expovariate(0.5), self, None)
```

Här definieras ett kösystem, i detta fall har det ingen buffert enbart en betjänare. I konstruktorn __init__ definieras följande:

- self.numberInQueue: anger hur många som finns i kösystemet, i detta fall är den alltid bara 0 eller 1
- self.measuredValues: en lista där man kan lagra antalet kunder i kösystemet, man lägger till ett värde varje gång en mätning görs
- self.sendTo: talar om vart en kund som är färdigbetjänad ska skickas
- self.numberBlocked: räknar hur många kunder som spärras. Om en kund kommer till kösystemet och betjänaren är upptagen (dvs self.numberInQueue == 1) så spärras kunden.
- self.numberServed: räknar hur många kunder som blir betjänade av kösystemet

I funktionen serviceTime drar man ett slumptal för betjäningstiden.

I funktionen treatSignal kan man ta hand om tre typer av signaler:

- ARRIVAL: en kund kommer. Då kollar man om betjänaren är ledig, om så är fallet så skickar kösysemet en signal av typen READY till sig själv som kommer fram efter en betjäningstid, ökar antalet kunder med 1, sparar undan informationen i signalen och ökar self.numberServed med 1. Om betjänaren inte är ledig, så ökar man self.numberBlocked med 1 och gör inte något mer.
- READY: en kund är färdigbetjänad. Då minskar man antalet kunder med 1 och skickar en signal ARRIVAL vidare. Den signalen innehåller self.info som anger när den färdigbetjänade kunden skapades av en generator.
- MEASUREMENT: en mätning ska göras. Antal kunder som finns i kösystemet (self.numberInQueue) läggs till listan self.measuredValues. Sedan skickar kösystemet en signal av typen MEASUREMENT till sig själv. När den tas omhand av kösystemet görs nästa mätning osv.

```
class sink(larger):

    def __init__(self):
        self.times = []
    def treatSignal(self, x, info):
        self.times.append(simTime - info)
```

I konstruktorn __init__ skapas en tom lista där tiderna som en kunderna har tillbringat i läggs in. När det kommer en signal till sink så beräknar man hur lång tid som har förflutit sedan kunden skapades och lägger till den tiden till listan

```
s = sink()
q = queue(s)
gen = generator(q)
```

Här skapas instanser av klasserna. Först en instans av sink, därefter en instans av queue. Konstruktorn för q får s som inparameter för q ska alltid skicka sina färdiga kunder till sink. På samma sätt så får konstruktorn för gen q som inparameter eftersom gen alltid ska skicka kunder som genereras till q.

```
send(READY, 0, gen, [])
send(MEASUREMENT, 10.0, q, [])
```

För att inte starta med en tom signalList så skickar vi två signaler. READY-signalen ser till att gen börjar generera kunder och MEASUREMENT-signalen ser till att q börjar göra mätningar av antalet kunder.

```
while simTime < stopTime:
    [simTime, signalType, dest, info] = heapq.heappop(signalList)
    dest.treatSignal(signalType, info)
```

Här är själva simuleringsloopen. Så länge som simuleringen inte är färdig (simTime < stopTime) så hämtar man den signal som har lägst värde på sin tidsstämpel från heapen, uppdaterar simTime till den signalens tidsstämpel och därefter tar man och anropar treatSignal i den instans av en klass som anges i destinationen.

```
print('Mean number in queue: ', np.mean(q.measuredValues))
print('Mean time in queue: ', np.mean(s.times))
print('Probability of blocking: ', q.numberBlocked/(q.numberServed +
q.numberBlocked))
```

Resultatet av simuleringen skrivs ut. Först skrivs medelvärdet av antalet kunder i kösystemet ut. Funktionen np.mean(…) beräknar medelvärdet av talen i listan q.measuredValues. På samma sätt beräknas medelvärdet av tiden som kunder har tillbringat i kösystemet ut i den andra utskriften. Slutligen beräknas sannolikheten att en kund spärras.