Random Walks: MITx 6.00.2x Lecture 6

Lucy Harlow

24/05/2021

```
Sys.setenv(RETICULATE_PYTHON = "/usr/local/bin/python3")
```

Lecture 5 concluded with a demonstration of simulating a random walk by hand. However, it's labour intensive after only a few steps, so a simulation is a good way to proceed.

The structure of the simulation will be as follows:

- Simulate one walk of k steps
- Simulate n such walks
- Report average distance from origin

The example I'll construct below is extremely close to Guttag's Location/Field/Drunk simulation from Lecture 6, but I'm making some modifications so that I make sure I'm understanding the code correctly.

Class Location

The location is a two-dimensional space, with only x and y coordinates. The horizontal and vertical movements – deltaX and deltaY – are floats.

```
class Location(object):
  def __init__(self, x, y):
    '''
    x and y are floats
    '''
    self.x = x
    self.y = y

  def move(self, deltaX, deltaY):
    '''
    deltaX and deltaY are floats
    '''
    return Location(self.x + deltaX, self.y + deltaY)

  def getX(self):
    return self.x

  def getY(self):
    return self.y

  def distFrom(self, other):
    ox = other.x
    oy = other.y
    xDist = self.x - ox
    yDist = self.x - ox
```

```
    return (xDist**2 + yDist**2)**0.5 # pythagoras

  def __str__(self):
    return '<' + str(self.x) + ', ' + str(self.y) + '>'
```

Class Dog Park

The key design decision here was to make the *location of the dog in the dog park* an attribute of the *dog park*, rather than an attribute of the *dog*. The reasoning is that it makes it possible to think more easily about how dogs relate to each other spatially – for example, the question of whether two dogs can occupy the same space in the dog park.

```
class dogPark(object):
  def __init__(self):
    self.dogs = {} # an empty dictionary

  def addDog(self, dog, loc):
    if dog in self.dogs:
      raise ValueError('Duplicate dog')
    else:
      self.dogs[dog] = loc

  def getLoc(self, dog):
    if dog not in self.dogs:
      raise ValueError('Dog not in park')
    return self.dogs[dog]

  def moveDog(self, dog):
    if dog not in self.dogs:
      raise ValueError('Dog not in park')
    xDist, yDist = dog.takeStep()
    currentLocation = self.dogs[dog]
    # use move method of Location to get new location
    self.dogs[dog] = currentLocation.move(xDist, yDist)
```

Notable aspects of class **Dog Park**:

- A mapping of dogs to locations
- Unbounded size
- Allows multiple dogs, with no constraints about how they relate to each other

Class Dog

```
class Dog(object):
  def __init__(self, name):
    self.name = name
  def __str__(self):
    return 'This dog is named ' + self.name
```

There's not much going on here! That's because it's not intended to be useful on its own. It's a **base class to be inherited**. We'll use this to create two subclasses of dog: the 'usual' dog, who wanders around at random, and the 'trash hound', who can smell the bins on the east side of the park, and wants to move towards them.

Two Kinds of Dogs

```
import random

class usualDog(Dog):
  def takeStep(self):
    stepChoices = [(0.0, 1.0), (0.0, -1.0), (1.0, 0.0), (-1.0, 0.0)]
    return random.choice(stepChoices)


class trashHound(Dog):
  def takeStep(self):
    stepChoices = [(0.0, 1.0), (0.0, -1.0), (1.1, 0.0), (-0.9, 0.0)]
    return random.choice(stepChoices)
```

The usual dog can take steps of exactly the same length north, south, east, and west respectively.

The trash hound, who wants to move east towards the trash, will move slightly further when travelling east, and slightly less far when travelling west.

Simulating a Single Walk

```
def walk(f, d, numSteps):
  '''
  Assumes that f is a Field, d is a Dog in f, and numSteps an int >= 0.
  Moves d numSteps times; returns the distance between the final location and the location at the start
  '''
  start = f.getLoc(d) # a method in dogPark
  for s in range(numSteps):
    f.moveDog(d) # a method in dogPark
  return start.distFrom(f.getLoc(d)) # distFrom, a method in Location
```

This is a nice, simple function, because we built the data abstractions above: this makes it obvious how to simulate a single walk.

Simulating Multiple Walks

This one's a little more complicated:

```
def simWalks(numSteps, numTrials, dClass):
  '''
  Assumes numSteps is an int >=0, numTrials is an int >-, dClass is a subclass of Dog.
  Simulates numTrials walks of numSteps steps each.
  Returns a list of the final distances for each trial.
  '''
  Luna = dClass()
  origin = Location(0, 0)
  distances = []
  for t in range(numTrials):
    f = Field()
    f.addDog(Luna, origin) # addDog a method from dogPark
    distances.append(round(walk(f, Luna, numTrials), 1))
  return distances
```