Introduction to Continuous Probability Distributions

In this lesson, we give an introduction to continuous probability distribution.

WE'LL COVER THE FOLLOWING

In the previous lesson, we described our first attempt of fixing `System.Random`:

- Make every method static and thread-safe
- Draw a clear distinction between crypto-strength and pseudo-random methods

---

Continuous Probability Distribution

Let's start by taking a step back and asking what we want when we're calling `NextInt` or `NextDouble` or whatever on a source of randomness. What we're saying is: we have a source of random `T`s for some type `T`, and the values produced will correspond to some probability distribution. We can express that very clearly in the C# type system:

```
public interface IDistribution<T>
{
  T Sample();
}
```

In this lesson, we are going to look at *continuous probability distributions*, which we will represent as `IDistribution<double>`. We'll look at integer distributions and other possible values for type parameter `T` in the upcoming lessons.

Now that we have an interface, what's the simplest possible implementation? We have a library that produces a uniform distribution of `double`s over the interval $[0.0, 1.0)$, which is called the **standard continuous uniform distribution**.

> Here, the notation $[x, y)$ is called "the half-open interval". It is standard notation. We notate real intervals as $(start, end)$. If a square parenthesis `[` OR `]` is used, that means that the value is *included*, and if a round parenthesis `(` OR `)` is used, that means the value is *excluded*.

Let's make a little singleton class for that:

```
using SCU = StandardContinuousUniform;
public sealed class StandardContinuousUniform : IDistribution<double>
{
  public static readonly StandardContinuousUniform
    Distribution = new StandardContinuousUniform();
  private StandardContinuousUniform() { }
  public double Sample() => Pseudorandom.NextDouble();
}
```

> For this course we are going to use pseudo-random numbers; feel free to sub in *crypto-strength* random numbers if you like it better.

> **Exercise:** Make this a non-singleton that takes a source of randomness as a parameter, and feel all fancy because now you're doing "dependency injection".

This implementation might seem trivial — and it is — but upon this simple foundation we can build a powerful abstraction.

Bridge between Distributions and LINQ

Now that we have a type that represents distributions, we would like to build a bridge between distributions and LINQ. The connection between probability

distributions and sequences should be pretty obvious; a **distribution** is a

thing that can produce an infinite sequence of values drawn from that distribution.

We will make a static class for our helper methods:

```
public static class Distribution
{
  public static IEnumerable<T> Samples<T>(
    this IDistribution<T> d)
  {
    while (true)
      yield return d.Sample();
  }
}
```

What does this give us? We instantly have access to all the representational power of the sequence operation library. Want a sum of twelve random doubles drawn from a uniform distribution? Then say that:

```
SCU.Distribution.Samples().Take(12).Sum()
```

> You might be thinking "if there is such a strong connection between distributions and infinite sequences then why not cut out the middleman by making `IDistribution<T>` extend the `IEnumerable<T>` interface directly, rather than requiring a `Samples()` helper method?" That's an excellent question; the answer will become clear in a few lessons.

Visualizing the Distribution

For testing and pedagogic purposes we'd like to be able to very quickly visualize a distribution from the console or in the debugger, so here are the extension methods:

```
public static string Histogram (
  this IDistribution <double> d, double low, double high) =>
    d.Samples().Histogram(low, high);
```

```
public static string Histogram (this IEnumerable <double> d, double low, d
```

```
ouble high)
{
  const int width = 40;
  const int height = 20;
  const int sampleCount = 100000;
  int[] buckets = new int[width];
  foreach(double c in d.Take(sampleCount))
  {
    int bucket = (int)(buckets.Length * (c - low) / (high - low));
    if (0 <= bucket && bucket < buckets.Length)
      buckets[bucket] += 1;
  }
  int max = buckets.Max();
  double scale = max < height ? 1.0 :((double)height) / max;
  return string.Join("",
    Enumerable.Range(0, height).Select(
      r => string.Join(“”, buckets.Select(
        b => b * scale > (height - r) ? ‘*’ : ‘ ‘)) + “\n”)) + new string(
‘-‘, width) + “\n”;
}
```

We can now do this:

```
SCU.Distribution.Histogram(0, 1)
```

and get the string:

```
**** ********** *    *   * * *******   *
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
****************************************
```

```
**************************************
```

This is the desired probability distribution histogram.

Extension methods and fluent programming are nice and all, but what else does this get us? Well, we can very concisely produce derived probability distributions. For example, suppose we want the normal distribution:

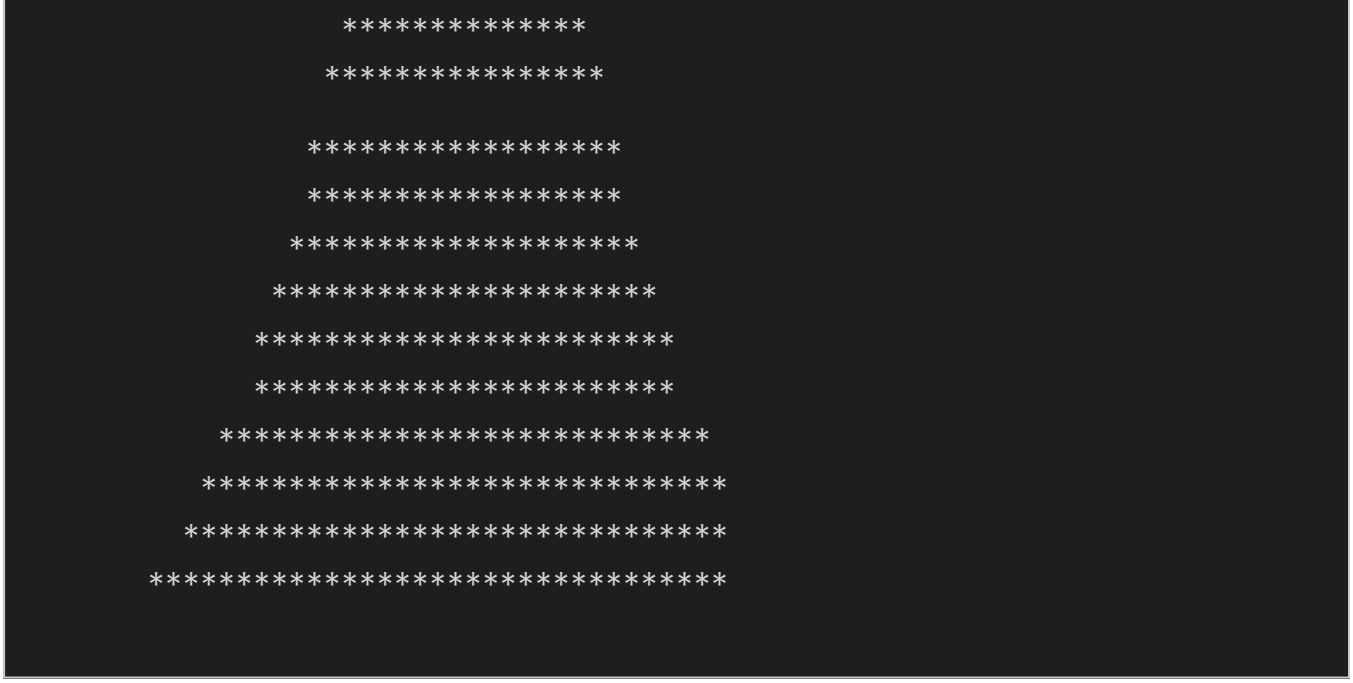
```
using static System.Math;
using SCU = StandardContinuousUniform;
public sealed class Normal : IDistribution<double>
{
  public double Mean { get; }
  public double Sigma { get; }
  public double μ => Mean;
  public double σ => Sigma;
  public readonly static Normal Standard = Distribution(0, 1);
  public static Normal Distribution(double mean, double sigma) => new Norm
al(mean, sigma);
  private Normal(double mean, double sigma)
  {
    this.Mean = mean;
    this.Sigma = sigma;
  }
  // Box - Muller method
  private double StandardSample() => Sqrt(-2.0 * Log(SCU.Distribution.Samp
le())) * Cos(2.0 * PI * SCU.Distribution.Sample());
  public double Sample() => μ + σ * StandardSample();
}
```

And of course we can graph it:

```
Normal.Distribution(1.0, 1.5).Histogram(-4, 4)
```

That will produce the following:

```
                    ***
                  ******
                 ********
                **********
                ***********
               ************
              **************
```



If you want a list with a hundred thousand normally distributed values with a particular mean and standard deviation, just ask for it:

```
Normal.Distribution(1.0, 1.5).Samples().Take(100000).ToList()
```

Hopefully, you can see how much more useful, readable and powerful this approach is, compared to messing around with `System.Random` directly. We want to move the level of abstraction of the code away from the mechanistic details of loops and lists. The code should be at the level of the business domain: distributions and population samples.

> **Exercise:** Try implementing the *Irwin-Hall distribution* (hint: it's already written in this lesson!)

> **Exercise:** Try implementing the *Gamma distribution* using the *Ahrens-Dieter algorithm*.

> **Exercise:** Try implementing the *Poisson distribution* as an `IDistribution<int>`.

Implementation

The code snippet below combines the concepts we have learned in this lesson.

Program.cs

BetterRandom.cs

Distribution.cs

Episode03.cs

Extensions.cs

IDistribution.cs

Normal.cs

Pseudorandom.cs

StandardCont.cs

```
using System;
using System.Linq;
namespace Probability
{
    using SCU = StandardContinuousUniform;
    static class Episode03
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 03\nThe sum of 12 random doubles:");
            Console.WriteLine(SCU.Distribution.Samples().Take(12).Sum());
            Console.WriteLine("A histogram of the SCU:");
            Console.WriteLine(SCU.Distribution.Histogram(0, 1));
            Console.WriteLine("A histogram of a Gaussian:");
            Console.WriteLine(Normal.Distribution(1.0, 1.5).Histogram(-4, 4));
        }
    }
}
```

In the next few lessons, we'll stop looking at continuous distributions and take a deeper look at the use cases for our new interface when the type argument represents a small set of discrete value, like `Booleans`, small `ints` or `enums`.