Introduction to Discrete Distributions

In this lesson, we will introduce discrete distributions.

WE'LL COVER THE FOLLOWING

In the previous lesson, we showed how we could use a simple interface to implement concepts like *"normal distribution"* or *"standard uniform distribution"*. For the next lessons, we'll leave continuous distributions aside and consider only **discrete distributions**. That is, we have some probability distribution where there are only a small, fixed number of outcomes. Flip a coin, roll three dice and add them together, that sort of thing. You can read up on its definition in the appendix chapter.

Discrete Distribution

Since we assume the number of possible outcomes of a discrete distribution is small. Therefore, we can enumerate them; the set of possible outcomes is, oddly enough, called the **support** of the distribution.

Some elements might be more likely than others; the ones that happen more often we say have more **weight** than the ones that happen less often. We want to be able to describe the weight of each member of a distribution easily. Also, we do not want to get off in the weeds of representing weights as doubles because double arithmetic has many pitfalls. For the first part of this course, we are going to assume that we can weight the members of a distribution by an integer. When we return to continuous distributions, we will consider the pros and cons of double weights further.

```
public interface IDiscreteDistribution<T> : IDistribution<T>
{
  IEnumerable<T> Support();
  int Weight(T t);
}
```

We are going to require that the weight function accept any value of type `T`. If the value is not in the support, the weight is zero, and if the value is in the support, the weight is a positive integer. We are going to further assume that the support and weights are small enough that the total weight fits into an integer.

> If we divide the weight function by the total weight (as a double) then we have the **probability mass function** of the discrete distribution.

Recall that we identified that System.Random.Next has a confusing interface because it is easy to think you are generating numbers from, say, 1 to 6, but actually, are generating numbers from 1 to 5. Let's make a better interface for this operation using our interface as follows:

```
using SCU = StandardContinuousUniform;
public sealed class StandardDiscreteUniform : IDiscreteDistribution<int>
{
  public static StandardDiscreteUniform Distribution(int min, int max)
  {
    if (min > max)
      throw new ArgumentException();
    return new StandardDiscreteUniform(min, max);
  }

  public int Min { get; }
  public int Max { get; }
  private StandardDiscreteUniform(int min, int max)
  {
    this.Min = min;
    this.Max = max;
  }

  public IEnumerable<int> Support() => Enumerable.Range(Min, 1 + Max - Min
```

```
);
  public int Sample() => (int)((SCU.Distribution.Sample() * (1.0 + Max - M
in)) + Min);
  public int Weight(int i) => (Min <= i && i <= Max) ? 1 : 0;
  public override string ToString() => $"StandardDiscreteUniform[{Min}, {M
ax}]";
}
```

Finally, we can now do things like `10d6` (`D&D` notation for *"sum ten six-sided die rolls"*) in a natural, fluent style. I would far prefer:

```
SDU.Distribution(1, 6).Samples().Take(10).Sum()
```

to

```
random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7) + ...
```

or the equivalent loop we typically see.

Reasons for Using Lower Bound Inclusive and UpperBound Exclusive

A common defense of "lower bound inclusive, upper bound exclusive" is that it is natural for programmers in C-family languages. It is easier to think of an array or list as indexed from 0 (inclusive) to the array length (exclusive). We would want to code things like:

```
myArray[Random.Next(0, myArray.Length)]
```

and not get confused and have to say `myArray.Length-1`.

Well, our answer to that (germane) critique is: once again, we're reasoning at the wrong level. If we want a probability distribution drawn from elements of an array, then don't mess around with array indexing and choosing the right random number and so on. That is writing code at the level of the mechanisms, not the business domain. Instead, make an object that represents a distribution of elements chosen from a given array and sample from it!

But we are getting ahead of ourselves; we'll get to that in an upcoming lesson.

Discrete Distribution Histogram

Previously we made a little “histogram string” helper extension method that is useful for quickly displaying results in a debugger or console. Let’s do the same for discrete distributions. Moreover, to make sure that we are getting the expected results, we are going to implement it by sampling rather than basing it on the support and the corresponding weights:

```
public static string Histogram<T>(this IDiscreteDistribution<T> d)
  => d.Samples().DiscreteHistogram();
```

```
public static string DiscreteHistogram<T>(this IEnumerable<T> d)
{
  const int sampleCount = 100000;
  const int width = 40;
  var dict = d.Take(sampleCount)
    .GroupBy(x => x)
    .ToDictionary(g => g.Key, g => g.Count());
  int labelMax = dict.Keys
  .Select(x => x.ToString().Length)
  .Max();

  Func<T, string> ToLabel = t =>
    t.ToString().PadLeft(labelMax);

  var sup = dict.Keys.OrderBy(ToLabel).ToList();
  int max = dict.Values.Max();
  double scale = max < width ? 1.0 : ((double)width) / max;

  Func<T, string> Bar = t =>
    new string('*', (int)(dict[t] * scale));

  return sup.Select(s => $"{ToLabel(s)}|{Bar(s)}").NewlineSeparated();
}
```

Let’s try this and roll `1d10`:

```
StandardDiscreteUniform.Distribution(1, 10).Histogram();
```

The histogram appears as we’d expect:

```
 1|****************************************
 2|****************************************
```

```
 3|**************************************
 4|**************************************
 5|**************************************
 6|**************************************
 7|**************************************
 8|**************************************
 9|**************************************
10|**************************************
```

Of course, sometimes we will want to see the weights directly:

```
public static string ShowWeights<T>(this IDiscreteDistribution<T> d)
{
  int labelMax = d.Support()
    .Select(x => x.ToString().Length)
    .Max();

  Func<T, string> ToLabel = t =>
    t.ToString().PadLeft(labelMax);

  return d.Support()
    .Select(s => $"{ToLabel(s)}:{d.Weight(s)}")
    .NewlineSeparated();
}
```

We seem to be doing pretty well here. Our implementations of each probability distribution are short and sweet, and each one provides a stone in the foundation for building more complex distributions.

Implementation

Let's have a look at the complete code for this lesson:

Program.cs

BetterRandom.cs

Distribution.cs

Episode04.cs

Extensions.cs

IDiscreteDistribution.cs

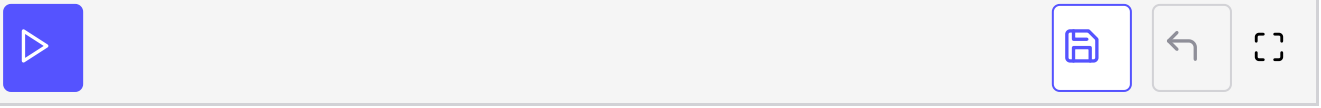
IDistribution.cs

Pseudorandom.cs

StandardCont.cs

StandardDiscrete.cs

```
using System;
using System.Linq;
namespace Probability
{
  using SDU = StandardDiscreteUniform;
  static class Episode04
  {
    public static void DoIt()
    {
      Console.WriteLine("Episode 04");
      Console.WriteLine("10d6:");
      Console.WriteLine(SDU.Distribution(1, 6).Samples().Take(10).Sum());
      Console.WriteLine("1d10:");
      Console.WriteLine(SDU.Distribution(1, 10).Histogram());
      Console.WriteLine("1d6:");
      Console.WriteLine(SDU.Distribution(1, 6).ShowWeights());
    }
  }
}
```



A distribution that, say, is over all the integers, or a billion integers, is also discrete, but in the next chapter, we want to look specifically at the easiest possible case: just a handful of discrete options.