```julia
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(Base.current_project())
    using Omega, Distributions, UnicodePlots, OmegaExamples
end
```

```
Activating project at `~/Documents/GitHub/Omega.jl/OmegaExamples`
```

The line between “reasoning” and “learning” is unclear in cognition. Just as reasoning can be seen as a form of conditional inference, so can learning: discovering persistent facts about the world (for example, causal processes or causal properties of objects). By saying that we are learning “persistent” facts we are indicating that there is something to infer which we expect to be relevant to many observations over time. Thus, we will formulate learning as inference in a model that (1) has a fixed latent value of interest, the hypothesis, and (2) has a sequence of observations, the data points.

When thinking about learning as inference, there are several key questions. First, what can be inferred about the hypothesis given a certain subset of the observed data? For example, in most cases, you cannot learn much about the weight of an object based on its colour. However, if there is a correlation between weight and colour – as is the case in many children’s toys – observing colour does allow you to learn about weight.

Second, what is the relationship between the amount of input (how much data we’ve observed) and the knowledge gained? In psychology, this relationship is often characterized with a learning curve, representing a belief as a function of amount of data. In general, getting more data allows us to update our beliefs. But some data, in some models, has a much bigger effect. In addition, while knowledge often changes gradually as data is accumulated, it sometimes jumps in non-linear ways; these are usually the most psychologically interesting predictions.

Example: Learning About Coins

As a simple illustration of learning, imagine that a friend pulls a coin out of her pocket and offers it to you to flip. You flip it five times and observe a set of all heads:

```
[H, H, H, H, H].
```

Does this seem at all surprising? To most people, flipping five heads in a row is a minor coincidence but nothing to get excited about. But suppose you flip it five more times and continue to observe only heads. Now the data set looks like this:

```
[H, H, H, H, H, H, H, H, H, H]
```

Most people would find this a highly suspicious coincidence and begin to suspect that perhaps their friend has rigged this coin in some way – maybe it’s a weighted coin that always comes up heads no matter how you flip it. This inference could be stronger or weaker, of course, depending on what you believe about your friend or how she seems to act; did she offer a large bet that you would flip more heads than tails? Now you continue to flip five more times and again observe nothing but heads – so the data set now consists of 15 heads in a row:

```
[H, H, H, H, H, H, H, H, H, H, H, H, H, H, H]
```

Regardless of your prior beliefs, it is almost impossible to resist the inference that the coin is a trick coin.

This *learning curve* reflects a highly systematic and rational process of conditional inference. For simplicity let's consider only two hypotheses, two possible definitions of coin, representing a fair coin and a trick coin that produces heads 95% of the time. A priori, how likely is any coin offered up by a friend to be a trick coin? Of course there is no objective or universal answer to that question, but for the sake of illustration let's assume that the *prior probability* of seeing a trick coin is 1 in a 1000, versus 999 in 1000 for a fair coin.

```
observed_data =  [1, 1, 1, 1, 1]
```

```julia
observed_data = [1, 1, 1, 1, 1]
```

```
fair_prior = 0.999
```

```julia
fair_prior = 0.999
```

```
fair (generic function with 1 method)
```

```julia
fair(fair_prior) = @~ Bernoulli(fair_prior)
```

```
coin (generic function with 1 method)
```

```julia
coin(i, ω, f) = ((@uid, i) ~ Bernoulli(f(ω) ? 0.5 : 0.95))(ω)
```

```
obs_fn (generic function with 1 method)
```

```julia
obs_fn(obs) = Variable(ω -> map(i -> coin(i, ω, fair(fair_prior)), 1:length(obs)))
```

```
fair_posterior (generic function with 1 method)
```

```julia
fair_posterior(obs) = fair(fair_prior) |ᶜ pw(==, obs_fn(obs), obs)
```

```
        ┌                                        ┐
  false ┤■ 23
   true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 977
        └                                        ┘
```

```julia
viz(randsample(fair_posterior(observed_data), 1000))
```

Try varying the number of flips and the number of heads observed. You should be able to reproduce the intuitive learning curve described above. Observing 5 heads in a row is not enough to suggest a trick coin, although it does raise the hint of this possibility: its chances are now a few percent, approximately 30 times the baseline chance of 1 in a 1000. After observing 10 heads in a row, the odds of trick coin and fair coin are now roughly comparable, although fair coin is still a little more likely. After seeing 15 or more heads in a row without any tails, the odds are now strongly in favour of the trick coin.

When exploring learning as a conditional inference, we are particularly interested in the dynamics of how inferred hypotheses change as a function of amount of data (often thought of as time the learner spends acquiring data). We can map out the trajectory of learning by plotting a summary of the posterior distribution as a function of the amount of observed data. Here we plot the expectation that the coin is fair in the above example:

```
true_weight = 0.9
```

```julia
true_weight = 0.9
```

```
observed_data_sizes =  [1, 3, 6]
```

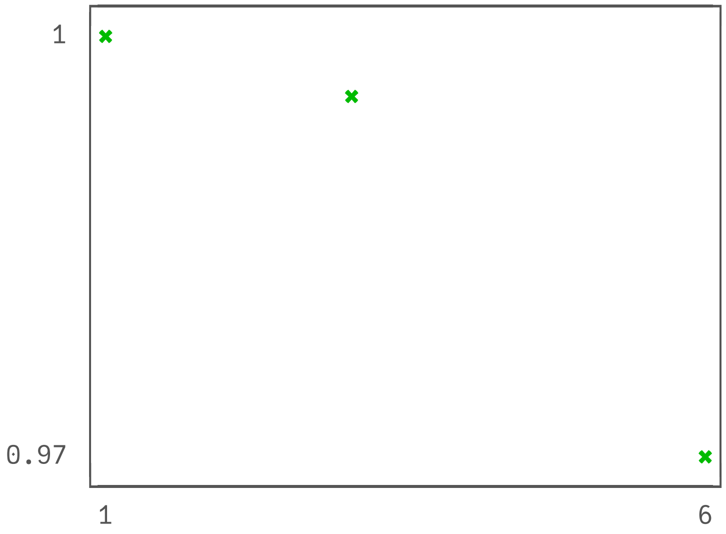
```julia
# observed_data_sizes = [1, 3, 6, 10, 20, 50, 100]
observed_data_sizes = [1, 3, 6]
```

```
estimates (generic function with 1 method)
```

```julia
estimates(n, ω) =
    fair_posterior(manynth((Bernoulli(true_weight)), 1:n)(ω))(ω)
```

```
[0.998, 0.995, 0.971]
```

```julia
begin
    p_estimates =
        map(n -> randsample(ω -> estimates(n, ω), 1000), observed_data_sizes)
    p_estimates = mean.(p_estimates)
end
```



```julia
scatterplot(observed_data_sizes, p_estimates, marker = :xcross)
```

Notice that different runs of this program can give quite different trajectories, but always end up in the same place in the long run. This is because the data set used for learning is different on each run. This is a feature, not a bug: real learners have idiosyncratic experience, even if they are all drawn from the same distribution. Of course, we are often interested in the average behavior of an ideal learner: we could average this plot over many randomly chosen data sets, simulating many different learners.

Study how this learning curve depends on the choice of `fair_prior`. There is certainly a dependence. If we set `fair_prior` to be 0.5, equal for the two alternative hypotheses, just 5 heads in a row are sufficient to favor the trick coin by a large margin. If `fair_prior` is 99 in 100, 10 heads in a row are sufficient. We have to increase `fair_prior` quite a lot, however, before 15 heads in a row is no longer sufficient evidence for a trick coin: even at `fair_prior = 0.9999`, 15 heads without a single tail still weighs in favor of the trick coin. This is because the evidence in favor of a trick coin accumulates exponentially as the data set increases in size; each successive head flips increases the evidence by nearly a factor of 2.

Learning is always about the shift from one state of knowledge to another. The speed of that shift provides a way to diagnose the strength of a learner's initial beliefs. Here, the fact that somewhere between 10 and 15 heads in a row is sufficient to convince most people that the coin is a trick coin suggests that for most people, the a priori probability of encountering a trick coin in this situation is somewhere between 1 in a 100 and 1 in $10,000$—a reasonable range. Of course, if you begin with the suspicion that any friend who offers you a coin to flip is liable to have a trick coin in his pocket, then just seeing five heads in a row should already make you very suspicious—as we can see by setting `fair_prior` to a value such as 0.9.

Independent and Exchangeable Sequences

Now that we have illustrated the kinds of questions we are interested in asking of learning models, let's delve into the mathematical structure of models for sequences of observations.

If the observations have nothing to do with each other, except that they have the same distribution, they are called *identically, independently distributed* (usually abbreviated to i.i.d.). For instance the values that come from calling flip are i.i.d. To verify this, let's first check whether the distribution of two `Bernoulli` in a sequence look the same (are "identical"):

```
gen_sequence =
  [(-3751694283739017014, 1)@Distributions.Bernoulli{Float64}(p=0.5), (-3751694283739017014,
```

```
gen_sequence = map(i -> ((@uid, i) ~ Bernoulli()), 1:2)
```

```
gen_sequence_samples =
  [[false, true], [false, true], [true, true], [false, true], [false, false], [true, true], [fal
```

```
gen_sequence_samples = randsample(ω -> mapf(ω, gen_sequence), 10000)
```

```
         ┌                                        ┐
   false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 5 014
    true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 4 986
         └                                        ┘
```

```
viz(map(x -> x[1], gen_sequence_samples))
```

```
         ┌                                        ┐
   false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 4 985
    true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 5 015
         └                                        ┘
```

```
viz(map(x -> x[2], gen_sequence_samples))
```

Now let's check that the first and second flips are independent, by conditioning on the first and seeing that the distribution of the second is unchanged:

```
test_independence (generic function with 1 method)
```

```julia
function test_independence(gen_sequence, val, ω)
    (gen_sequence[1] |ᶜ (gen_sequence[1] .== val))(ω)
    gen_sequence[2](ω)
end
```

```
         ┌                                        ┐
   false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 494
    true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 506
         └                                        ┘
```

```julia
viz(randsample(ω -> test_independence(gen_sequence, true, ω), 1000))
```

```
         ┌                                        ┐
   false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 476
    true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 524
         └                                        ┘
```

```julia
viz(randsample(ω -> test_independence(gen_sequence, false, ω), 1000))
```

It is easy to build other i.i.d.s in Omega. For instance, here is an extremely simple model for the words in a sentence:

```
words =  ["chef", "omelet", "soup", "eat", "work", "bake", "stop"]
```

```julia
words = ["chef", "omelet", "soup", "eat", "work", "bake", "stop"]
```

```
probs =  [0.00325, 0.48633, 0.07891, 0.06754, 0.19741, 0.13879, 0.02777]
```

```julia
probs = [0.00325, 0.48633, 0.07891, 0.06754, 0.19741, 0.13879, 0.02777]
```

```
words_class (generic function with 1 method)
```

```julia
words_class(i, ω) = (pget(words) ∘ ((@uid, i) ~ Categorical(probs)))(ω)
```

```
 ["work", "work", "bake", "omelet", "omelet", "omelet", "omelet", "work", "omelet", "omelet"]
```

```julia
randsample(manynth(words_class, 1:10))
```

In this example the different words are indeed independent: you can show as above (by conditioning) that the first word tells you nothing about the second word. However, constructing sequences in this way it is easy to accidentally create a sequence that is not entirely independent. For instance:

```
probs_dep (generic function with 1 method)
```

```julia
probs_dep(ω)  = ((@~ Bernoulli())(ω) ?
                [0.00325, 0.48633, 0.07891, 0.06754, 0.19741, 0.13879, 0.02777] :
                [0.36994, 0.12965, 0.02783, 0.41318, 0.02392, 0.01599, 0.01949])
```

```
words_dep_class (generic function with 1 method)
```

```julia
words_dep_class(i, ω) = (pget(words) ∘ ((@uid, i) ~ Categorical(probs_dep(ω))))(ω)
```

```
 ["eat", "eat", "chef", "eat", "eat", "omelet", "chef", "chef", "omelet", "eat"]
```

```julia
randsample(manynth(words_dep_class, 1:10))
```

While the sequence looks very similar, the words are not independent: learning about the first word tells us something about the `probs_dep`, which in turn tells us about the second word. Let's show this in a slightly simpler example:

```
gen_sequence_ (generic function with 1 method)
```

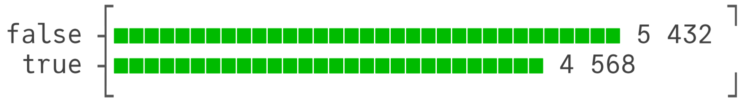
```
gen_sequence_(i, ω) = ((@uid, i) ~ Bernoulli((@~ Bernoulli())(ω) ? 0.2 : 0.7))(ω)
```

```
gen_sequence_dep =  [1@gen_sequence_, 2@gen_sequence_]
```

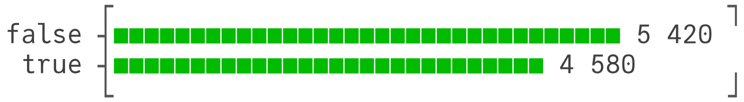
```
gen_sequence_dep = map(i -> i ~ gen_sequence_, 1:2)
```

```
gen_sequence_dep_samples =
 [[true, true], [true, true], [true, false], [true, false], [true, false], [true, false], [true
```

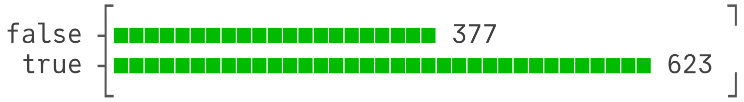
```
gen_sequence_dep_samples = randsample(ω -> mapf(ω, gen_sequence_dep), 10000)
```



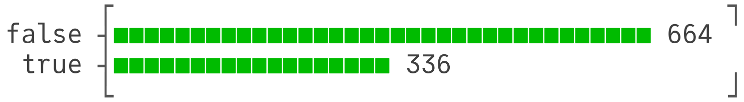
```
viz(map(x -> x[1], gen_sequence_dep_samples))
```



```
viz(map(x -> x[2], gen_sequence_dep_samples))
```



```
viz(randsample(ω -> test_independence(gen_sequence_dep, true, ω), 1000))
```



```
viz(randsample(ω -> test_independence(gen_sequence_dep, false, ω), 1000))
```

Conditioning on the first value tells us something about the second. This model is thus not i.i.d., but it does have a slightly weaker property: it is *exchangeable*, meaning that the probability of a sequence of values remains the same if permuted into any order. When modeling learning it is often reasonable that the order of observations doesn't matter—and hence that the distribution is exchangeable.

It turns out that exchangeable sequences can always be modeled in the form used for the last example: *de Finetti's theorem* says that, under certain technical conditions, any exchangeable sequence can be represented in terms of some `latent_prior` distribution and observation function `f`.

Example: Polya's Urn

A classic example is Polya's urn: Imagine an urn that contains some number of white and black balls. On each step we draw a random ball from the urn, note its color, and return it to the urn along with *another* ball of that color. Here is this model in Omega:

```
urn_seq (generic function with 1 method)
```

```julia
function urn_seq(urn, num_samples, ω)
    if num_samples == 0
        return empty(urn)
    else
        ball = ((@uid, num_samples) ~ Omega.UniformDraw(urn))(ω)
        return vcat(ball, urn_seq(vcat(urn, ball), num_samples - 1, ω))
    end
end
```

```
                ┌                                         ┐
['b', 'b', 'b'] ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 238
['b', 'b', 'w'] ┤■■■■■■■■■■■■ 87
['b', 'w', 'b'] ┤■■■■■■■■■■■ 81
['b', 'w', 'w'] ┤■■■■■■■■■■ 75
['w', 'b', 'b'] ┤■■■■■■■■■■■ 81
['w', 'b', 'w'] ┤■■■■■■■■■■■■ 87
['w', 'w', 'b'] ┤■■■■■■■■■■■■■ 93
['w', 'w', 'w'] ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 258
                └                                         ┘
```

```julia
viz(randsample(ω -> string(urn_seq(['b', 'w'], 3, ω)), 1000))
```

Polya's urn is an examples of a "rich get richer" dynamic, which has many applications for modeling the real world. Examining the distribution on sequences, it appears that this model is exchangeable—permutations of a sequence all have the same probability (e.g., `['b', 'b', 'w']`, `['b', 'w', 'b']`, `['w', 'b', 'b']` have the same probability; `['b', 'w', 'w']`, `['w', 'b', 'w']`, `['w', 'w', 'b']` do too). (Challenge: Can you prove this mathematically?)

Because the distribution is exchangeable, we know that there must be an alterative representation in terms of a latent quantity followed by independent samples. The de Finetti representation of this model is:

```
urn_de_Finetti (generic function with 1 method)
```

```julia
function urn_de_Finetti(urn, num_samples, ω)
    num_white = count(x -> x =='w', urn)
    num_black = length(urn) - num_white
    latent_prior = ((@uid, num_samples) ~ Beta(num_white, num_black))(ω)
    map(i -> ((i ~ Bernoulli(latent_prior))(ω) ? 'b' : 'w'), 1:num_samples)
end
```

```
                ┌                                         ┐
['b', 'b', 'b'] ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 248
['b', 'b', 'w'] ┤■■■■■■■■■■■■ 86
['b', 'w', 'b'] ┤■■■■■■■■■■■■■ 99
['b', 'w', 'w'] ┤■■■■■■■■■ 64
['w', 'b', 'b'] ┤■■■■■■■■■■■ 78
['w', 'b', 'w'] ┤■■■■■■■■■■■ 81
['w', 'w', 'b'] ┤■■■■■■■■■■■ 85
['w', 'w', 'w'] ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 259
                └                                         ┘
```

```julia
viz(randsample(ω -> string(urn_de_Finetti(['b', 'w'], 3, ω)), 1000))
```

We sample a shared latent parameter – in this case, a sample from a Beta distribution – generating the sequence samples independently given this parameter. We obtain the same distribution on sequences of draws. (Challenge: show mathematically that these two representations give the same distribution.)

Ideal learners

Recall that we aimed to formulate learning as inference in a model that has a fixed latent value of interest and a sequence of observations. We now know that this will be possible anytime we are willing to assume the data are exchangeable.

Many Bayesian models of learning are formulated in this way. We often write this in the pattern of Bayes' rule:

```
hypothesis = prior(ω)
obs_fn(datum) = ...uses hypothesis...
hypothesis |ᶜ obs_fn ==ₚ observed_data
```

The `prior` samples a hypothesis from the hypothesis space. This distribution expresses our prior knowledge about how the process we observe is likely to work, before we have observed any data. The function `obs_fn` captures the relation between the `hypothesis` and a single `datum`. (The marginal probability function for `obs_fn` is called the *likelihood*. Sometimes `obs_fn` itself is colloquially called the likelihood, too.)

Overall this setup of prior, likelihood, and a sequence of observed data (which implies an exchangeable distribution on data!) describes an *ideal learner*.

Example: Subjective Randomness

What does a random sequence look like? Is 00101 more random than 00000? Is the former a better example of a sequence coming from a fair coin than the latter? Most people say so, but notice that if you flip a fair coin, these two sequences are equally probable. Yet these intuitions about randomness are pervasive and often misunderstood: In 1936 the Zenith corporation attempted to test the hypothesis the people are sensitive to psychic transmissions. During a radio program, a group of psychics would attempt to transmit a randomly drawn sequence of ones and zeros to the listeners. Listeners were asked to write down and then mail in the sequence they perceived. The data thus generated showed no systematic effect of the transmitted sequence—but it did show a strong preference for certain sequences (Goodfellow, 1938). The preferred sequences included 00101, 00110, 01100, and 01101.

Griffiths and Tenenbaum (2001) suggested that we can explain this bias if people are considering not the probability of the sequence under a fair-coin process, but the probability that the sequence would have come from a fair process as opposed to a non-uniform (trick) process:

```
is_fair = -5589724309656828534@Distributions.Bernoulli{Float64}(p=0.5)
```

```
is_fair = @~ Bernoulli()
```

```
coin (generic function with 2 methods)
```

```julia
coin(i, ω) = ((@uid, i) ~ Bernoulli(is_fair(ω) ? 0.5 : 0.2))(ω)
```

```
rand_coins (generic function with 1 method)
```

```julia
rand_coins(seq) = manynth(coin, 1:length(seq))
```

```
is_fair_dist (generic function with 1 method)
```

```julia
is_fair_dist(seq) = is_fair |ᶜ pw(==, rand_coins(seq), seq)
```

To check if 00000 is fair:

```
seq1 =  [false, false, false, false, false]
```

```julia
seq1 = [false, false, false, false, false]
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 910
   true ┤■■■ 90
        └                                        ┘
```

```julia
viz(randsample(is_fair_dist(seq1), 1000))
```

To check if 00101 is fair:

```
seq2 =  [false, false, true, false, true]
```

```julia
seq2 = [false, false, true, false, true]
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■■■■■■■■■ 407
   true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 593
        └                                        ┘
```

```julia
viz(randsample(is_fair_dist(seq2), 1000))
```

This model posits that when considering randomness people are more concerned with distinguishing a “truly random” generative process from a trick process. How do these inferences depend on the amount of data? Explore the learning trajectories of this model.

Learning a Continuous Parameter

The previous examples represent perhaps simple cases of learning. Typical learning problems in human cognition or AI are more complex in many ways. For one, learners are almost always confronted with more than two hypotheses about the causal structure that might underlie their observations. Indeed, hypothesis spaces for learning are often infinite. Countably infinite hypothesis spaces are encountered in models of learning for domains traditionally considered to depend on “discrete” or “symbolic” knowledge; hypothesis spaces of grammars in language acquisition are a canonical example. Hypothesis spaces for learning in domains traditionally considered more “continuous”, such as perception or motor control, are typically uncountable and parametrized by one or more continuous dimensions. In causal learning, both discrete and continuous hypothesis spaces typically arise. (In statistics, making conditional inferences over continuous hypothesis spaces given data is often called *parameter estimation*.)

We can explore a basic case of learning with continuous hypothesis spaces by slightly enriching our coin flipping example. Suppose instead of simply flipping a coin to determine which of two coin weights to use, we can choose any coin weight between 0 and 1. The following program computes conditional inferences about the weight of a coin drawn from a prior distribution described by the `Uniform` dsitribution, conditioned on a set of observed flips.

```
obs_data =  [1, 1, 1, 1, 1]
```

```
obs_data = [1, 1, 1, 1, 1]
```

```
coin_weight = 4123385804682217859@StdUniform{Float64}()
```

```
coin_weight = @~StdUniform{Float64}()
```

```
coin_ = #1 (generic function with 1 method)
```

```
coin_ = Bernoulli(coin_weight)
```

```
evidence (generic function with 1 method)
```

```
evidence(obs) = pw(==, manynth(coin_, 1:length(obs)), obs)
```

```
weight_posterior (generic function with 1 method)
```

```
weight_posterior(obs_data) = coin_weight |ᶜ evidence(obs_data)
```

```
              ┌                                        ┐
   [0.3, 0.4) ┤ 2
   [0.4, 0.5) ┤■ 8
   [0.5, 0.6) ┤■■ 32
   [0.6, 0.7) ┤■■■■■ 63
   [0.7, 0.8) ┤■■■■■■■■■■■■ 152
   [0.8, 0.9) ┤■■■■■■■■■■■■■■■■■■■■■■ 285
   [0.9, 1.0) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 458
              └                                        ┘
                              Frequency
```

```
viz(randsample(weight_posterior(obs_data), 1000))
```

Experiment with different data sets, varying both the number of flips and the relative proportion of heads and tails. How does the shape of the conditional distribution change? The location of its peak reflects a reasonable “best guess” about the underlying coin weight. It will be roughly equal to the proportion of heads observed, reflecting the fact that our prior knowledge is basically uninformative; a priori, any value of `coin_weight` is equally likely. The spread of the conditional distribution reflects a notion of confidence in our beliefs about the coin weight. The distribution becomes more sharply peaked as we observe more data, because each flip, as an independent sample of the process we are learning about, provides additional evidence of the process’s unknown parameters.

We can again look at the learning trajectory in this example:

```
estimates_ (generic function with 1 method)
```

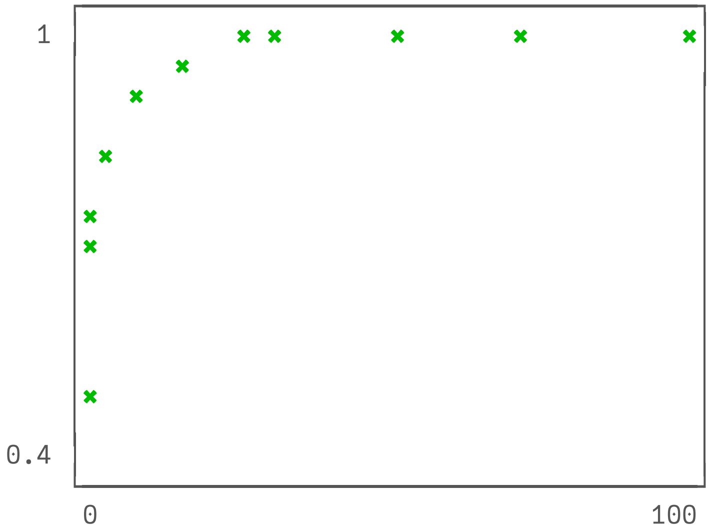
```
estimates_(n) = weight_posterior(ones(Int64, n))
```

```
obs_data_sizes =  [0, 1, 2, 4, 8, 16, 25, 30, 50, 70, 100]
```

```
obs_data_sizes = [0,1,2,4,8,16,25,30,50,70,100]
```

```
e =
  [0.491652, 0.6914, 0.755117, 0.826335, 0.880737, 0.947666, 0.965179, 0.970294, 0.981974, 0.9
```

```
e = map(n -> mean(randsample(estimates_(n), 100)), obs_data_sizes)
```



```
scatterplot(obs_data_sizes, e, marker = :xcross)
```

It is easy to see that this model doesn't really capture our intuitions about coins, or at least not in everyday scenarios. Imagine that you have just received a quarter in change from a store – or even better, taken it from a nicely wrapped-up roll of quarters that you have just picked up from a bank. Your prior expectation at this point is that the coin is almost surely fair. If you flip it 10 times and get 7 heads out of 10, you'll think nothing of it; that could easily happen with a fair coin and there is no reason to suspect the weight of this particular coin is anything other than 0.5. But running the above query with uniform prior beliefs on the coin weight, you'll guess the weight, in this case, is around 0.7. Our hypothesis generating function needs to be able to draw `coin_weight` not from a uniform distribution, but from some other function that can encode various expectations about how likely the coin is to be fair, skewed towards heads or tails, and so on.

One option is the Beta distribution. The Beta distribution takes parameters `α` and `β`, which describe the prior toward `true` and `false`. (When `α` and `β` are integers they can be thought of as *prior* observations.)

```
pseudo_counts =   (α = 10, β = 10)
```

```
pseudo_counts = (α = 10, β = 10)
```

```
coin_weight_ = 2974629498627146804@Distributions.Beta{Float64}(α=10.0, β=10.0)
```

```
coin_weight_ = @~ Beta(pseudo_counts...)
```

```
coin_beta = #1 (generic function with 1 method)
```

```
coin_beta = Bernoulli(coin_weight_)
```

```
evidence_beta (generic function with 1 method)
```

```
evidence_beta(obs) = pw(==, manynth(coin_beta, 1:length(obs)), obs)
```

```
weight_posterior_beta (generic function with 1 method)
```

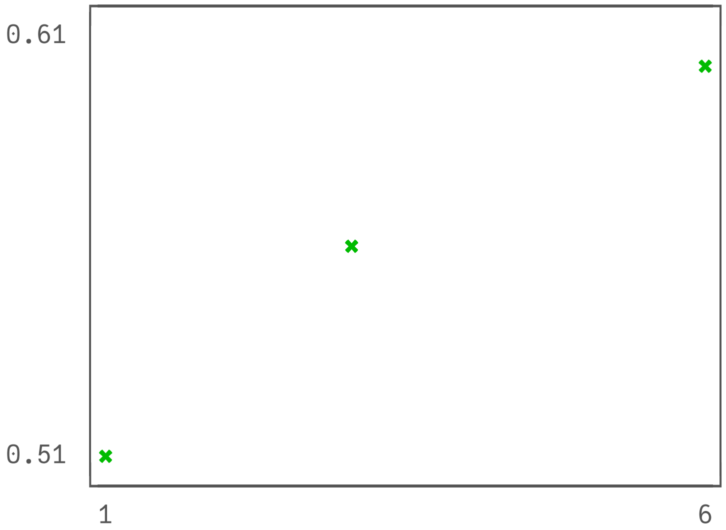
```
weight_posterior_beta(obs) = coin_weight_ |ᶜ evidence_beta(obs)
```

```
estimates_beta (generic function with 1 method)
```

```julia
estimates_beta(n) = weight_posterior_beta(ones(Int64, n))
```

```
e_ =  [0.5163, 0.560598, 0.601237]
```

```julia
e_ = map(n -> mean(randsample(estimates_beta(n), 100)), observed_data_sizes)
```



```julia
scatterplot(observed_data_sizes, e_, marker = :xcross)
```

We are getting closer, in that learning is far more conservative. In fact, it is too conservative: after getting heads 100 times in a row, most humans will conclude the coin can only come up heads. The model, in contrast, still expects the coin to come up tails around 10% of the time.

We can of course decrease our priors α and β to get faster learning, but then we will just go back to our earlier problem. We would like instead to encode in our prior the idea that fair coins (probability 0.5) are much more likely than even moderately unfair coins.

A More Structured Hypothesis Space

The following model explicitly builds in the prior belief that fair coins are likely, and that all unfair coins are equally likely as each other:

```
is_fair_ = -2382836693537082149@Distributions.Bernoulli{Float64}(p=0.999)
```

```julia
is_fair_ = @~ Bernoulli(0.999)
```

```
real_weight =
ifelseₚ(-2382836693537082149@Distributions.Bernoulli{Float64}(p=0.999), 0.5, 62584807432293
```

```julia
real_weight = ifelse.(is_fair_, 0.5, @~ StdUniform{Float64}())
```

```
coin_human_like (generic function with 1 method)
```

```julia
coin_human_like(i, ω::Ω) = ((@uid, i...) ~ Bernoulli(real_weight))(ω)
```

```
evidence_human_like (generic function with 1 method)
```

```julia
evidence_human_like(obs) = Variable(ω ->  all(isapprox.(manynth(coin_human_like,
1:length(obs))(ω), obs, atol=0.01)))
```

```
weight_posterior_human_like (generic function with 1 method)
```

```julia
weight_posterior_human_like(obs) = real_weight |ᶜ evidence_human_like(obs)
```

```
estimates_human_like (generic function with 1 method)
```

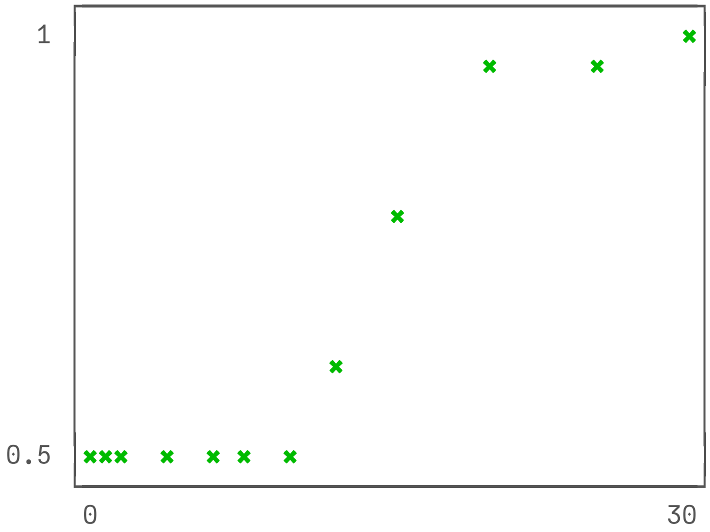
```julia
estimates_human_like(n) = weight_posterior_human_like(ones(Int64, n))
```

```
d_sizes =  [0, 1, 2, 4, 6, 8, 10, 12, 15, 20, 25, 30]
```

```julia
d_sizes = [0,1,2,4,6,8,10,12,15,20,25,30]
```

```
exp =
 [0.5, 0.5, 0.501898, 0.503188, 0.513632, 0.510575, 0.525862, 0.623282, 0.772828, 0.95073, 0.
```

```julia
exp = map(n -> mean(randsample(estimates_human_like(n), 100)), d_sizes)
```



```julia
scatterplot(d_sizes, exp, marker = :xcross)
```

This model stubbornly believes the coin is fair until around 10 successive heads have been observed. After that, it rapidly concludes that the coin can only come up heads. The shape of this learning trajectory is much closer to what we would expect for humans. This model is a simple example of a *hierarchical prior* which we explore in detail in a later chapter.

Example: Estimating Causal Power

Modeling beliefs about coins makes for clear examples, but it's obviously not a very important cognitive problem. However, many important cognitive problems have a remarkably similar structure.

For instance, a common problem for cognition is *causal learning*: from observed evidence about the co-occurrence of events, attempt to infer the causal structure relating them. An especially simple case that has been studied by psychologists is *elemental causal induction*: causal learning when there are only two events, a potential cause C and a potential effect E. Cheng and colleagues have suggested assuming that C and background effects can both cause E, with a noisy-or interaction. Causal learning then becomes an example of parameter learning, where the parameter is the "causal power" of C to cause E:

```
cp = 14822008039552515310StdUniform{Float64}()
```

```julia
cp = @~ StdUniform{Float64}() # Causal power of C to cause E
```

```
b = 40100571144671785380StdUniform{Float64}()
```

```julia
b = @~ StdUniform{Float64}() # Background probability of E
```

```
BitVector: [false, false]
```

```julia
randsample(pw(.&, manynth(Bernoulli(cp), 1:2), [false, false]))
```

```
obs_function (generic function with 1 method)
```

```julia
function obs_function(data)
    cp_ = manynth(Bernoulli(cp), 1:length(data.C))
    b_ = manynth(Bernoulli(b), 1:length(data.C))
    pw(==, pw(.|, pw(.&, cp_, data.C), b_), data.E)
end
```

```
data =  (C = [true, true, false, true], E = [true, true, false, true])
```

```julia
data = (C = [true, true, false, true], E = [true, true, false, true])
```

```
                 ┌                                         ┐
   [0.0, 0.2) ┤■■■ 4
   [0.2, 0.4) ┤■■■■■■■■ 11
   [0.4, 0.6) ┤■■■■■■■■ 11
   [0.6, 0.8) ┤■■■■■■■■■■■■■■■■■■■■■ 27
   [0.8, 1.0) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 47
                 └                                         ┘
                                  Frequency
```

```julia
viz(randsample(cp |ᶜ obs_function(data), 100))
```

Experiment with this model: when does it conclude that a causal relation is likely (high `cp`)? Does this match your intuitions? What role does the background rate `b` play? What happens if you change the functional relationship in `obs_function`?