Tight Genes:

Intro to Genetic Algorithms

by Dave Aronson
T.Rex-2023@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson
github.com/CodosaurusLLC/tight-genes

Hei, Oslo!

(Hello Oslo!)

Image: standard emoji

Speaker notes

Hi, osLO!

Jeg er Dave Aronson,

Image: me speaking at JSConf Hawai'i 2020!

(I'm Dave Aronson,)

Speaker notes

Yai ar Dave Aronson,

T. Rex fra Codosaurus,

(the T. Rex of Codosaurus,)

Image: my company logo!

Speaker notes

Tay Rex fra Codosowroos,

og jeg fløy hit

🛫 ● ● ● 🛬

(and I flew here)

Image: standard emoji

Speaker notes

oh yai floy heet

på kjæledyret mitt pterodactyl

(on my pet pterodactyl)

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

Speaker notes

po KYELihDEEReh mit PeroduckTEEL

for å lære deg om

(to teach you about)

Image: standard emoji

Speaker notes

for oh LAARa dai om

Genetiske Algoritmer.

(Genetic Algorithms.)

Image: https://pixabay.com/vectors/genetic-testing-gene-panel-genetics-2316642

Speaker notes

GeNEtiskEH AlgoREETmer?

Men . . .

(But . . .)

Image: standard emoji

Speaker notes

M'n . . .

jeg skal gjøre det på engelsk.

(I will do it in English.)

Image: standard emoji

Yai skuhl YOR-uh deuh po en-YILSK.

Mainly because you’ve just heard almost all the Norwegian I speak.

So what are genetic algorithms anyway? They are . . .

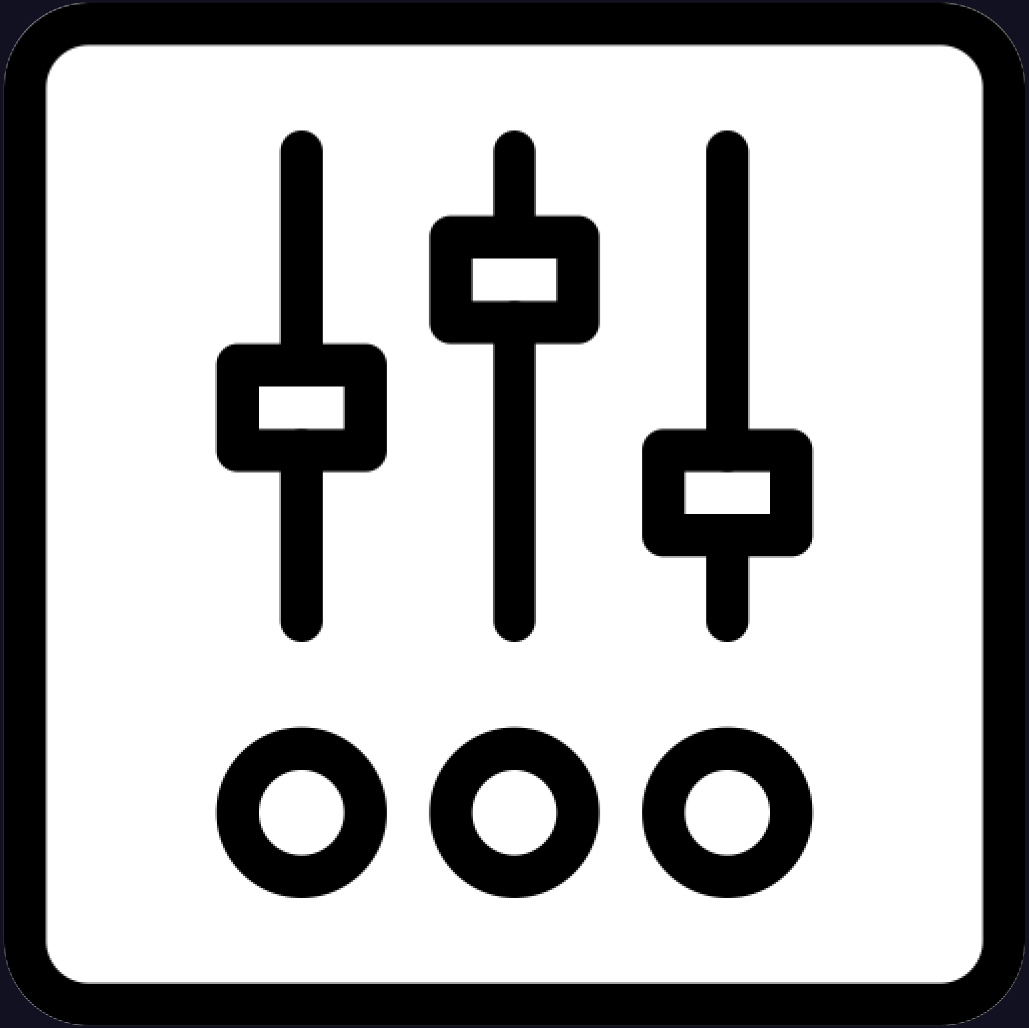


Image: https://icon-icons.com/download/49925/PNG/512/

Speaker notes

optimization meta-heuristics,

WAT?!

Speaker notes

which is fancy-talk for . . .

Optimization Meta-Heuristic:

category of shortcuts
to find
"good enough" solutions
(ideally the best,
but OK if not).

Speaker notes

. . . types of shortcuts to find a good solution to a problem, ideally the best, but we usually have to settle for something "good enough", due to constraints like time or money.

There are many kinds of optimization heuristics, but what makes genetic algorithms unique is that (as you may have guessed from the name) they are inspired by . . .

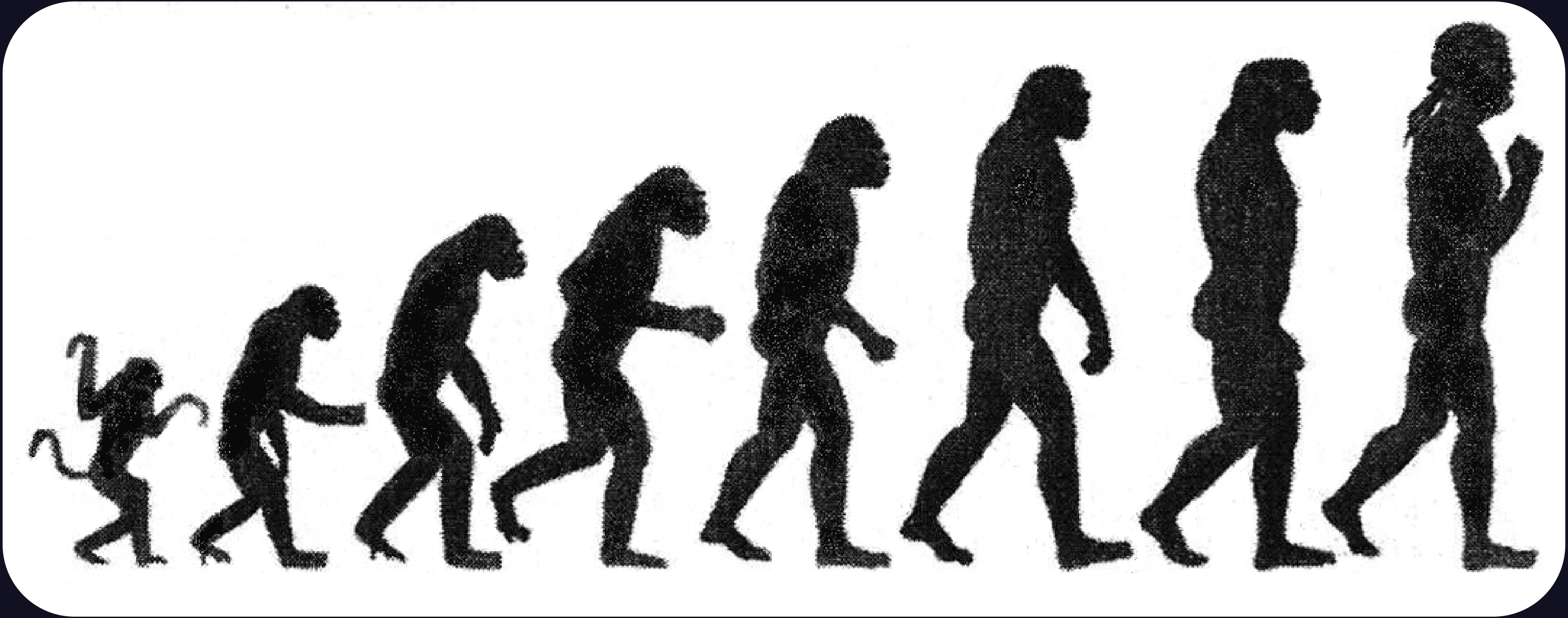
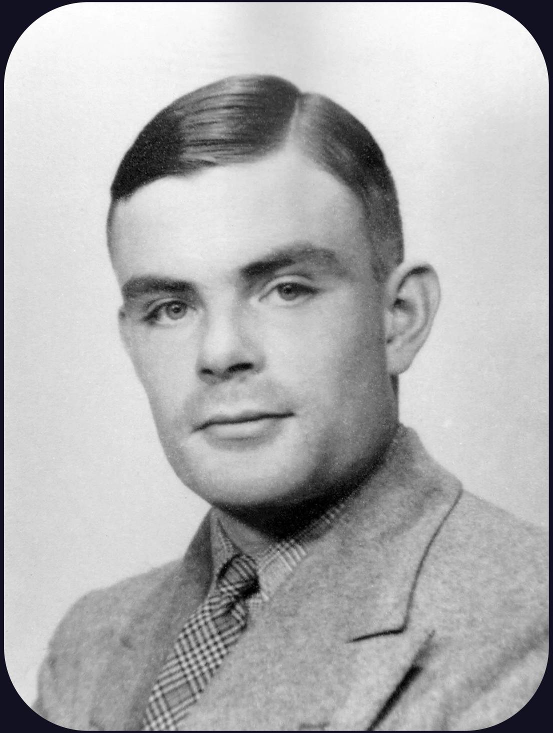


Image: https://commons.wikimedia.org/wiki/File:Darwin-chart.PNG

Speaker notes

. . . real-world biological evolution, mainly the principles of survival of the fittest, random combination of different sets of genes into one new set, and random mutation.

The history goes back to 1950, when . . .



Alan Turing

Image: https://cdn.britannica.com/81/191581-050-8C0A8CD3/Alan-Turing.jpg

Speaker notes

. . . Alan Turing, as in Turing Test, Turing Machine, Turing Completeness, and so on, proposed a "learning machine" in which the mechanism of learning would be similar to evolution. Nothing much came of that, nor of genetic algorithms in general... for a few decades. But then they finally got a bit of traction. The first commercial product based on genetic algorithms, a mainframe toolkit for . . .

Image: https://pxhere.com/en/photo/267871

Speaker notes

. . . industrial processes, came out in the late 1980's, from General Electric. In 1989 a genetic algorithm toolkit called Evolver came out for PCs. These days, MATLAB has a few genetic algorithm tools built-in, and many programming languages have genetic algorithm libraries available. However, the actual uses of genetic algorithms remain mostly obscure, used by companies in their internal industrial processes, logistics, scheduling, and so on.

But once in a while, they get used for something more interesting. Most famously, in 2005 . . .

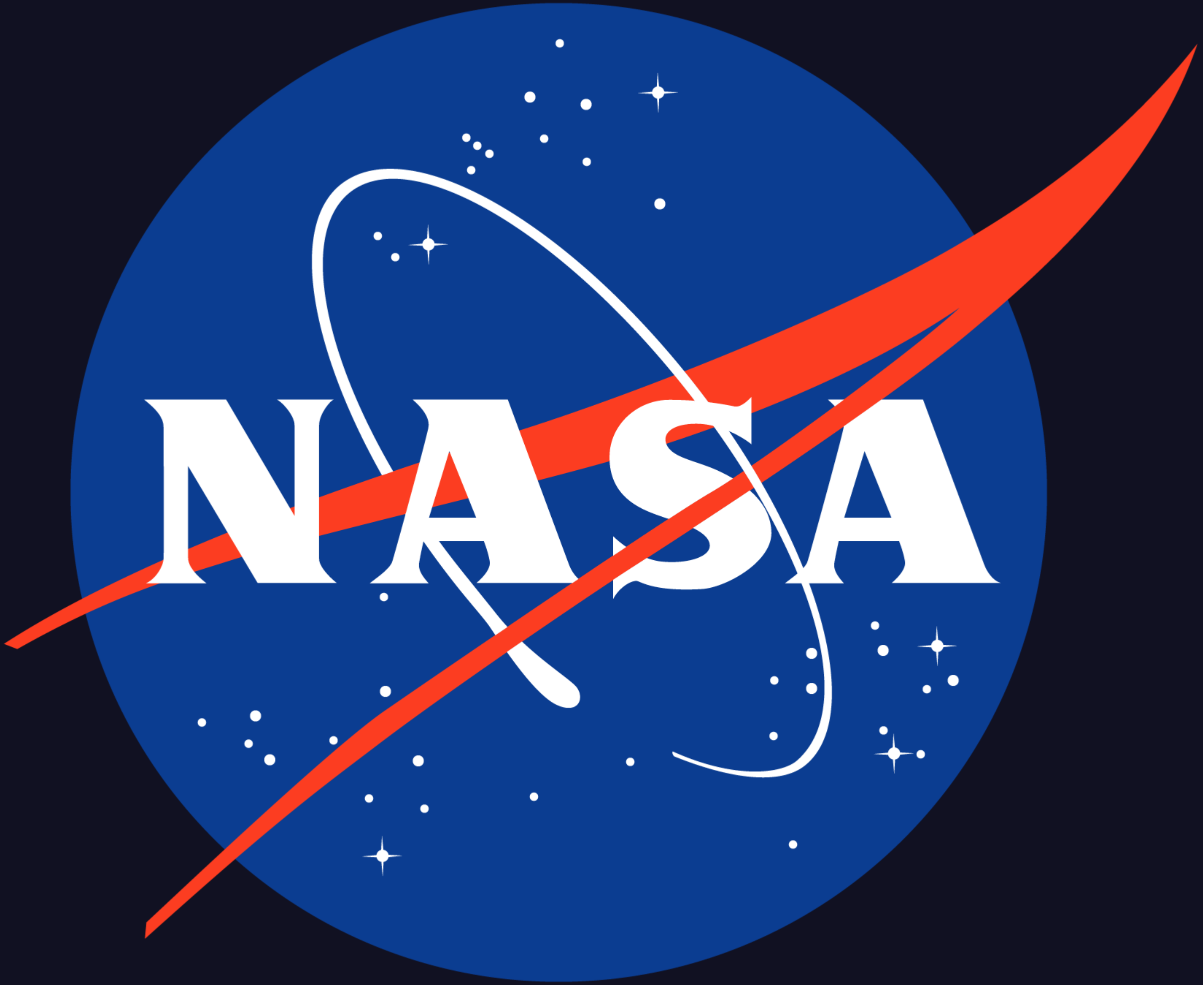
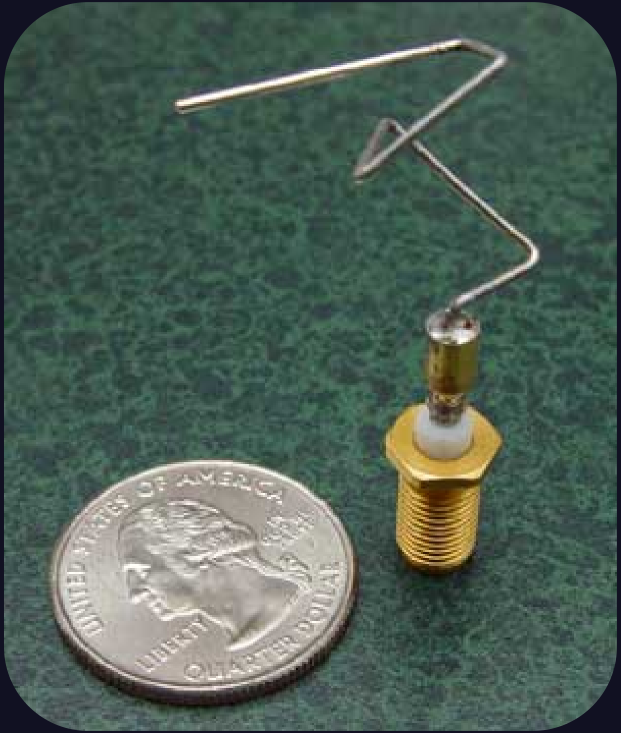


Image: https://www.nasa.gov/sites/default/files/thumbnails/image/nasa-logo-web-rgb.png

Speaker notes

. . . NASA used a genetic algorithm to design an . . .



ST5 satellite antenna, & quarter for scale

Image: https://www.jpl.nasa.gov/nmp/st5/IMAGES/st5-antenna.jpg

Speaker notes

. . . antenna for the ST5 series of satellites, launched in 2006. (No, that's not just a paperclip that got all bent up by someone sitting bored and fidgeting, in a meeting.) The NASA Jet Propulsion Laboratory website says: "Its unusual shape is expected because most human antenna designers would never think of such a design." And that is one of the great advantages of this approach. These algorithms are much less hampered by expectations of similarity to past solutions. However, today we'll only be looking at much simpler problem domains, for the sake of making understandable demos.

So how do genetic algorithms work? They consist of a simple series of steps:

Initialize

Speaker notes

First, we create an initial population of candidates. In Genetic Algorithm lingo, these are called "chromosomes", but since most living beings contain many chromosomes in each and every cell, I don't like that term, I think it leads to confusion, so I'm just going to say "candidates". I've also heard them called individuals, solutions, or phenotypes, to use an actual genetic term, but most people don't know that word, so I'll skip that one too.

The next step is to . . .

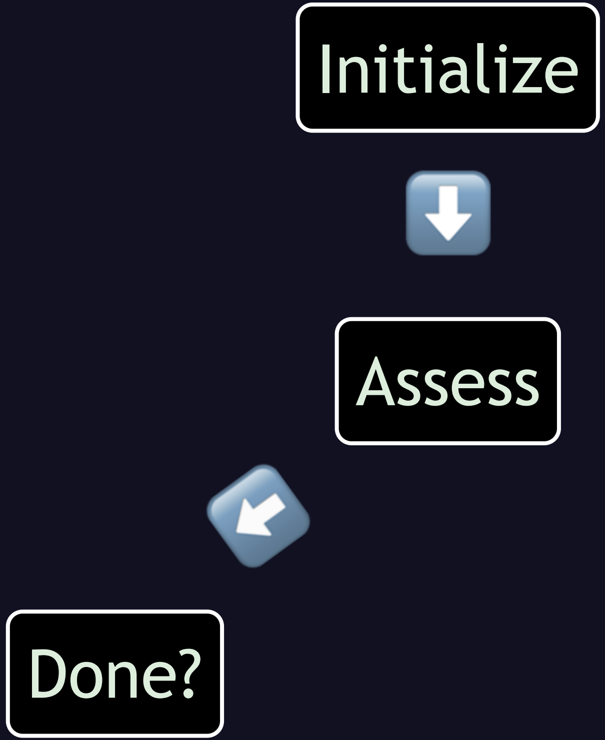
Initialize

⬇️

Assess

Speaker notes

. . . assess the "fitness" of each candidate, according to whatever criteria we want to apply. We do it here mainly because it supplies the data usually used in the next step, which is to ask, are we . . .



Initialize

⬇

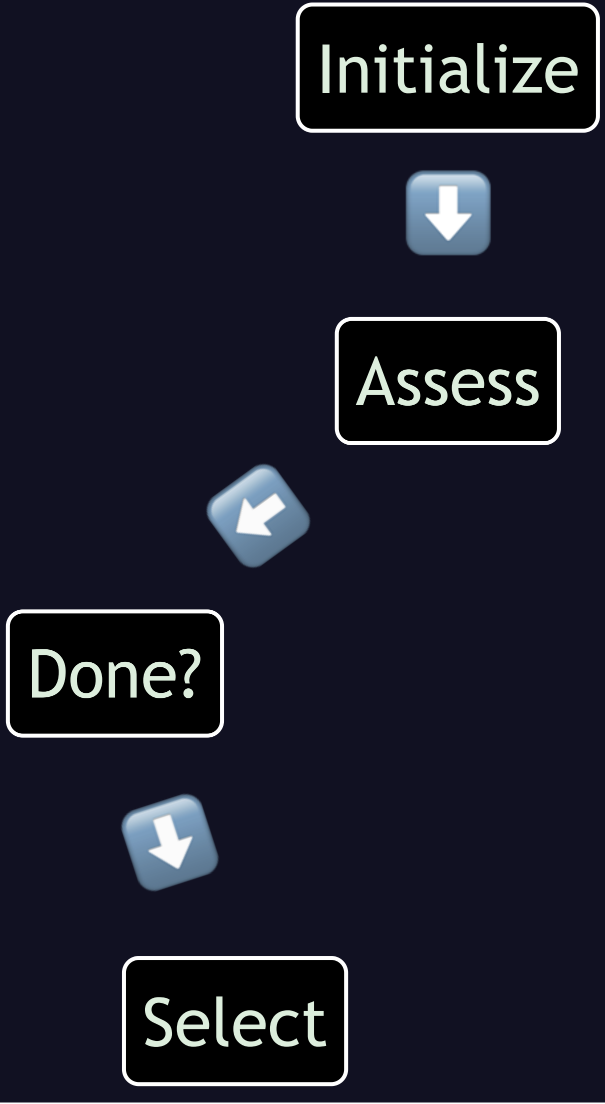
Assess

↙

Done?

Speaker notes

. . . done yet? This is usually based on the fitness, but could be based on other criteria, and we'll discuss some of those later, or a combination. If we're not done, then next we . . .



Initialize

↓

Assess

↙

Done?

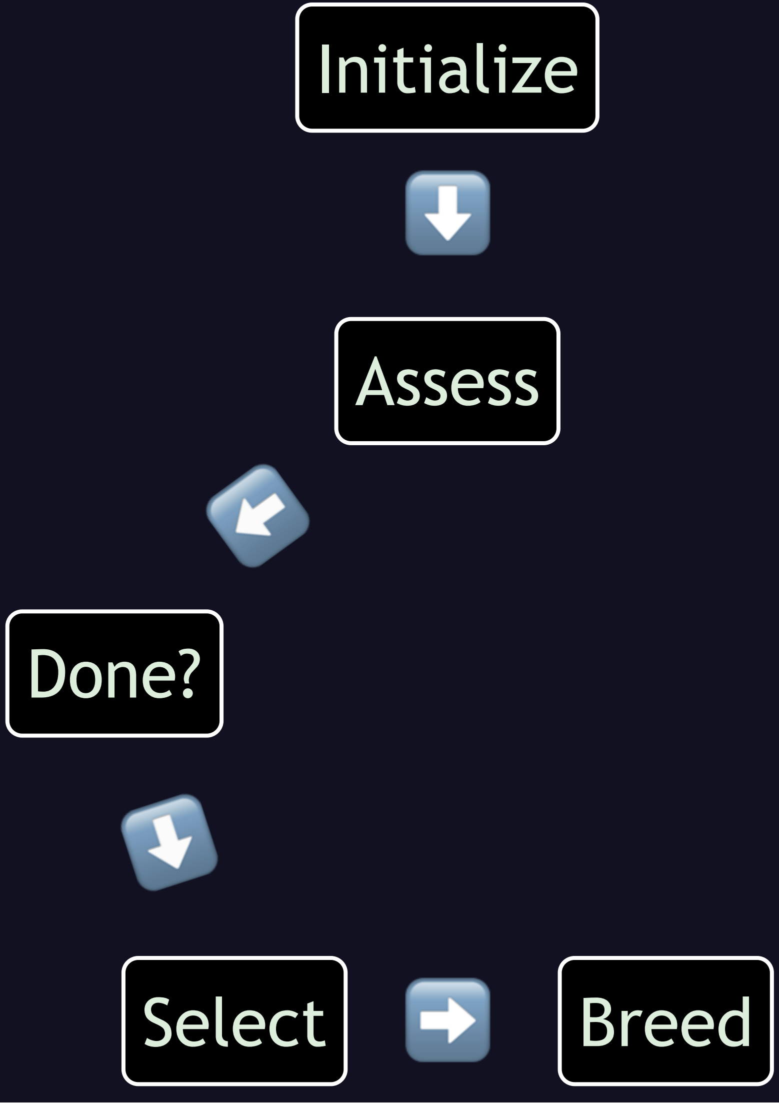
↓

Select

Speaker notes

. . . select some candidates to breed the next generation. This is also usually based on the fitness, to simulate survival of the fittest.

After that, as you may have guessed, we use those candidates we just selected . . .



Initialize

Assess

Done?

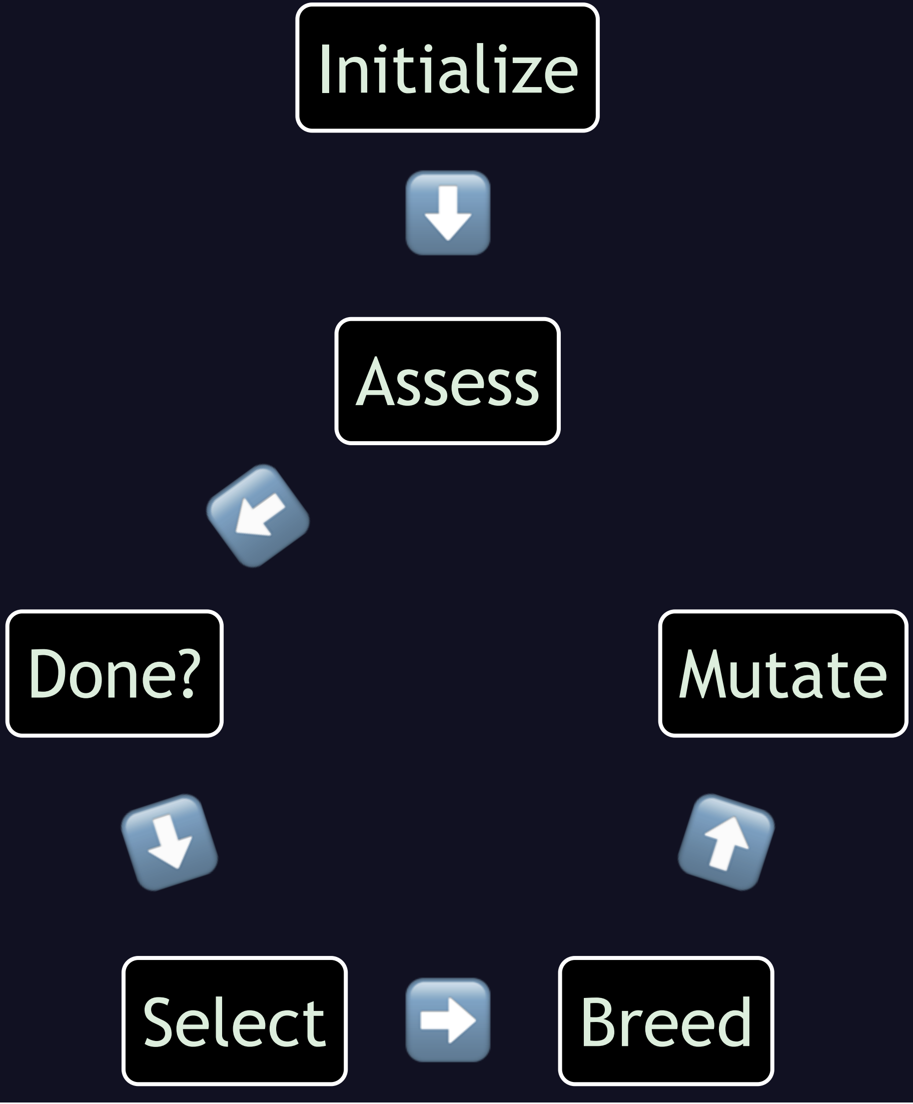
Select

Breed

Speaker notes

. . . to breed a new population. Some of the previous population, especially the fittest ones, may be carried over into this new population, but usually not.

Next is a very important but easily forgotten step, which is to . . .



Initialize

Assess

Done?

Select

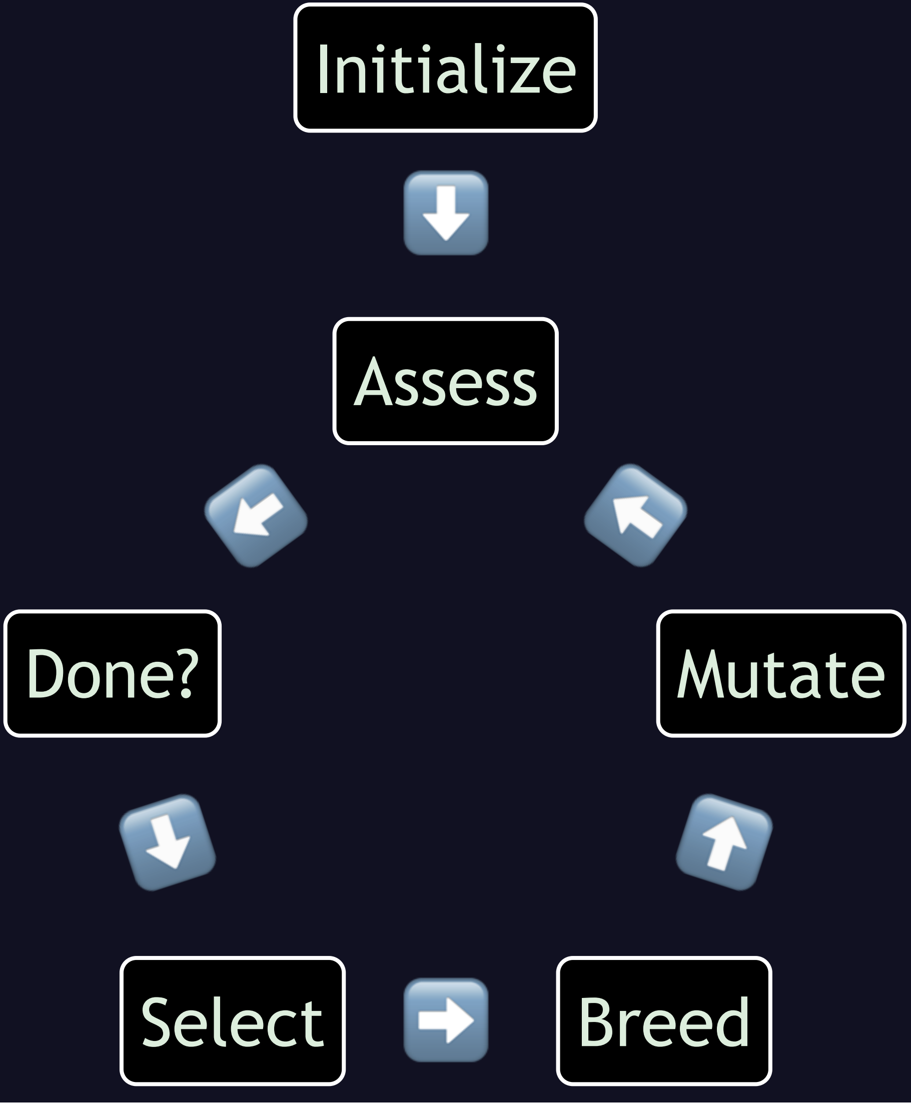
Breed

Mutate

Speaker notes

. . . mutate those new candidates, so that we get some more diversity in the gene pool.

Finally, we . . .



Initialize

Assess

Done?

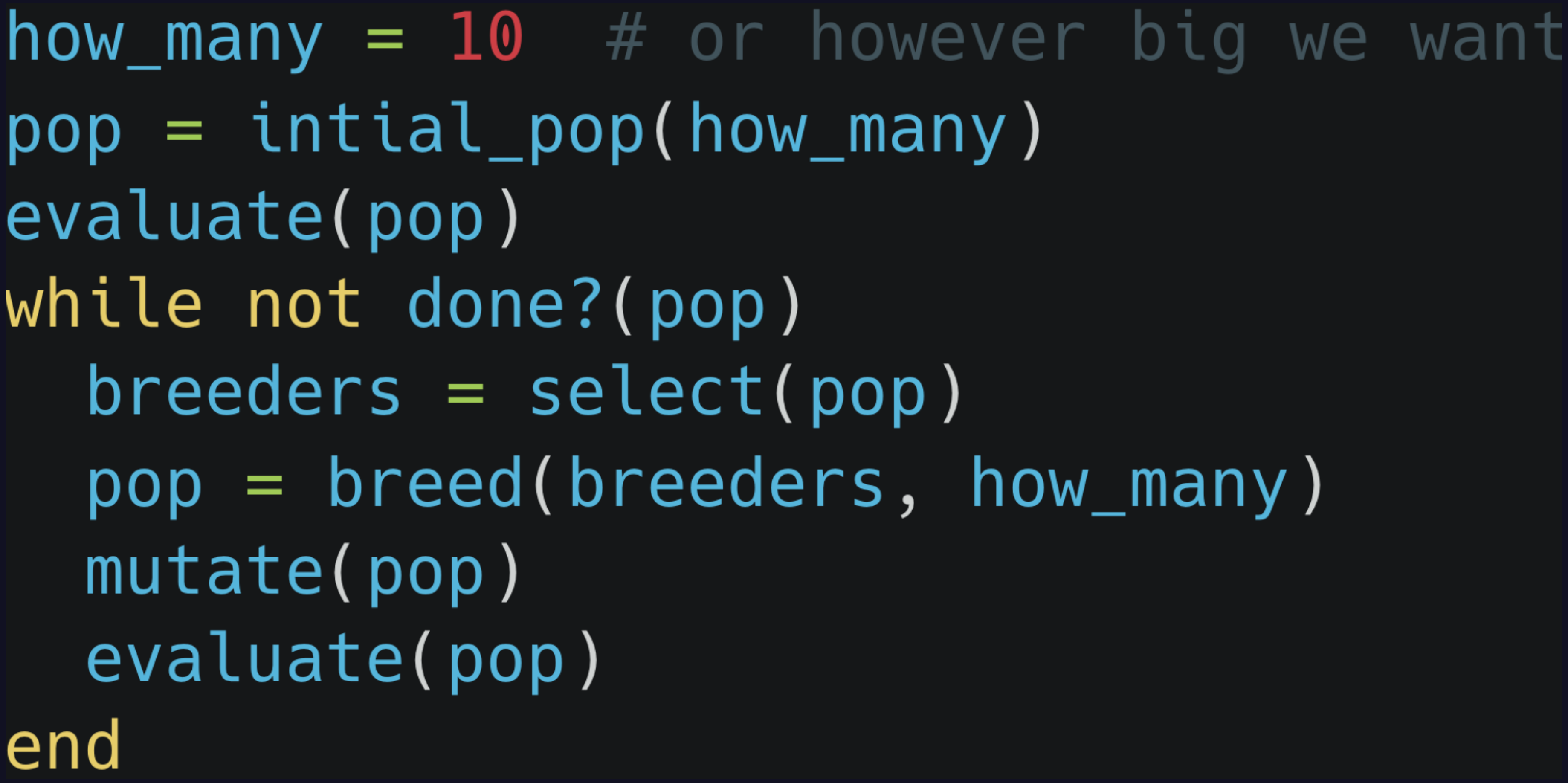
Select

Breed

Mutate

Speaker notes

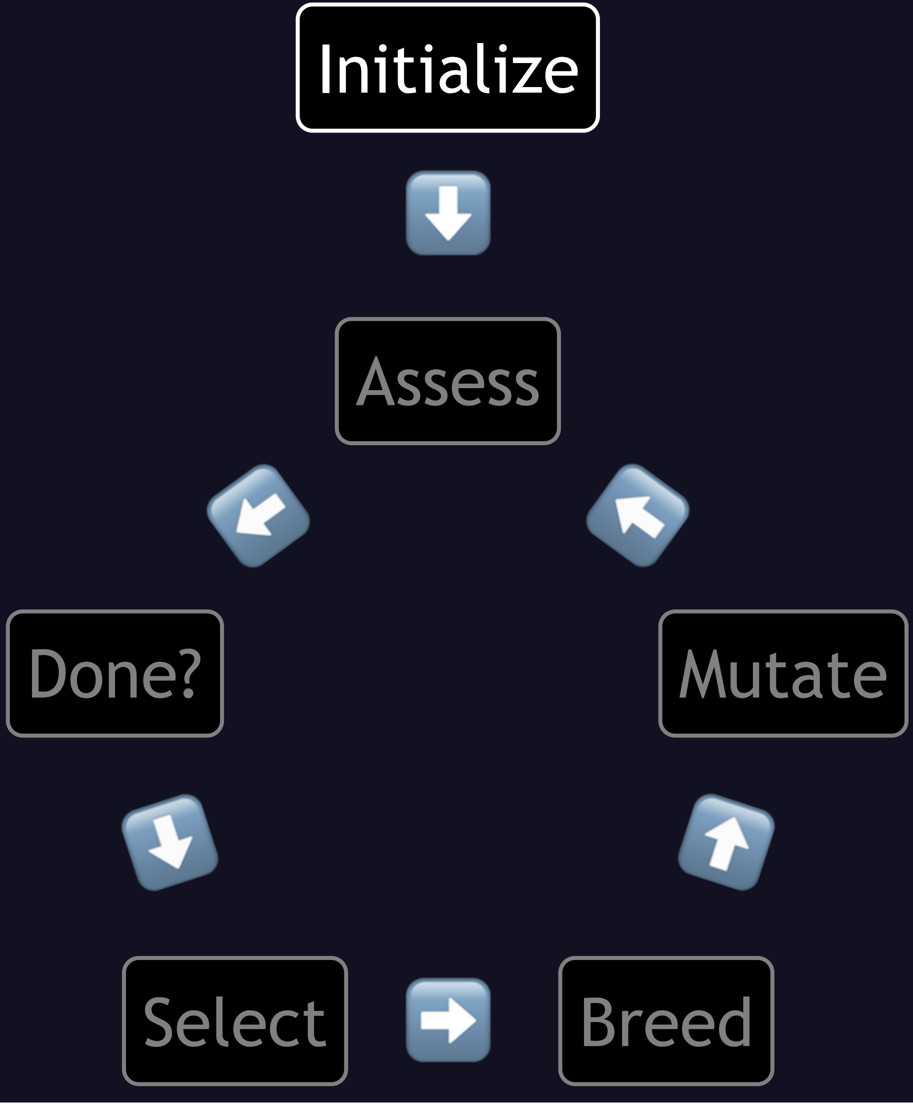
. . . go back to step 2, assessing their fitness. This sequence could be represented at a high level with some rather simple code, like so:



```
how_many = 10  # or however big we want
pop = intial_pop(how_many)
evaluate(pop)
while not done?(pop)
  breeders = select(pop)
  pop = breed(breeders, how_many)
  mutate(pop)
  evaluate(pop)
end
```

Speaker notes

. . . in Ruby. Now let's take a closer look at what goes on in each step, by working through an example.



Initialize

Assess

Done?

Mutate

Select

Breed

Speaker notes

First we create an initial population of candidates. But what is a candidate, and how do we create one? These are different solutions to some problem, usually represented as different instances of the same data structure. They could be any data structure we want, so long as we can evaluate their fitness, combine old ones to make a new one, and mutate them. The simplest common type of candidate is . . .

01001000

01100101

01101100

01101100

01101111

00100000

01110111

01101111

01110010

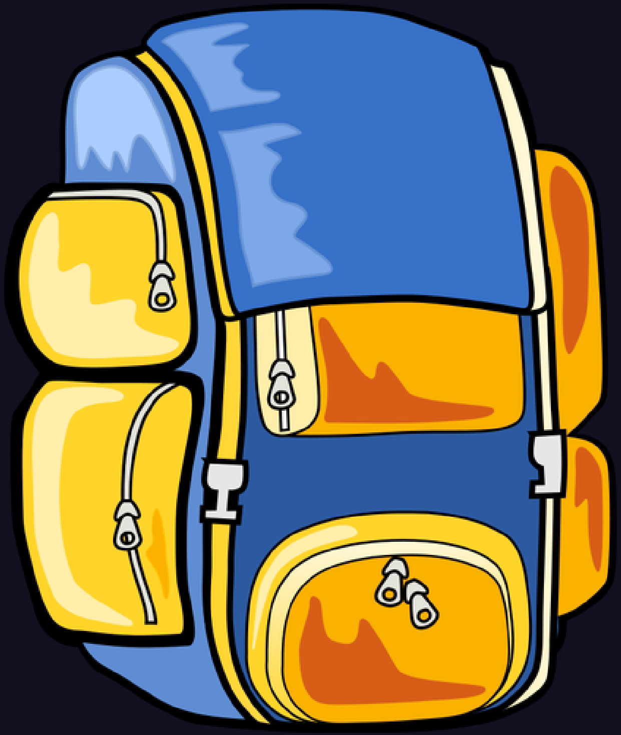
01101100

01100100

00100001

Speaker notes

. . . a simple string of bits. This will do fine for candidates that consist of a simple series of yes/no decisions. This may sound simplistic, but there is a huge class of problems that boil down to this, called . . .



Knapsack / Rucksack / Backpack / Whatever!

Image: https://publicdomainvectors.org/en/free-clipart/Rucksack-vector-image/9910.html

Speaker notes

. . . knapsack problems, which is a category of constrained resource allocation problems. The canonical example is that you have a knapsack -- or rucksack, backpack, or whatever you call it -- and many things you want to carry in it, but they won't all fit, or the total weight is more than you can carry, or some such similar constraint, or combination of constraints. So you want to find the combination of items, that will fit the constraints, and has the maximum value. That could be the literal cash value, as we'll see in a moment, or something more metaphorical. To look at a concrete example, suppose we know . . .

Image: standard emoji

Speaker notes

. . . a farmer, with a smallish truck, and he needs to decide what to take to market each week. And on this farm he has . . .

Images: standard emoji

Speaker notes

. . . some cows. (EIEIO!) So among the things he can take to market are:

Images: standard emoji plus https://www.rawpixel.com/image/6130389/

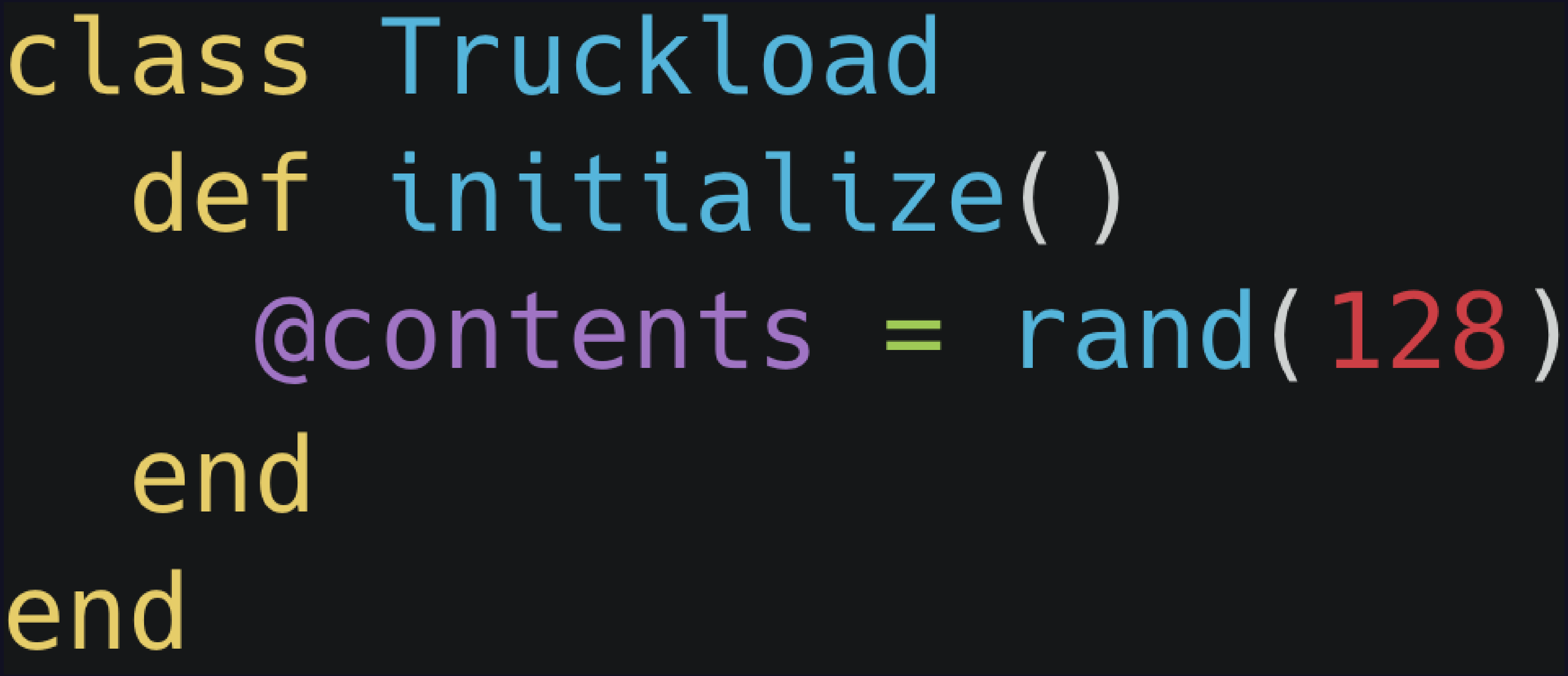
Speaker notes

. . . cows, milk, cheese, butter, ice cream, meat, and leather. For the sake of simplicity, we won't differentiate between between price and profit, nor dairy versus meat cows, and he can only take a set amount of each item, and always has that amount on hand. His truck has room to take all the items, but it can only carry so much weight, so that's our constraint. His choices are as follows:

What	Unit	Qty	Pounds	Value
Cow	cow	1	1,500	$2,000
Milk	1-gal jug	200	1,720	$800
Cheese	5-lb wheel	200	1,000	$12,000
Butter	1-lb block	1,000	1,000	$3,000
Ice Cream	1-gal tubs	200	1,000	$2,000
Meat	side	4	1,280	$8,000
Leather	hide	20	1,100	$6,000
TOTAL WEIGHT			8,600	

Speaker notes

You don't need to remember all that, just notice that it totals 8,600 pounds. But, his truck's suspension can only handle two tons, or 4,000 pounds. Let's see what happens if we use a genetic algorithm to determine a "good enough" truckload. First we need a way to represent each candidate. In code, we could represent them as a class, and create one randomly, like so:

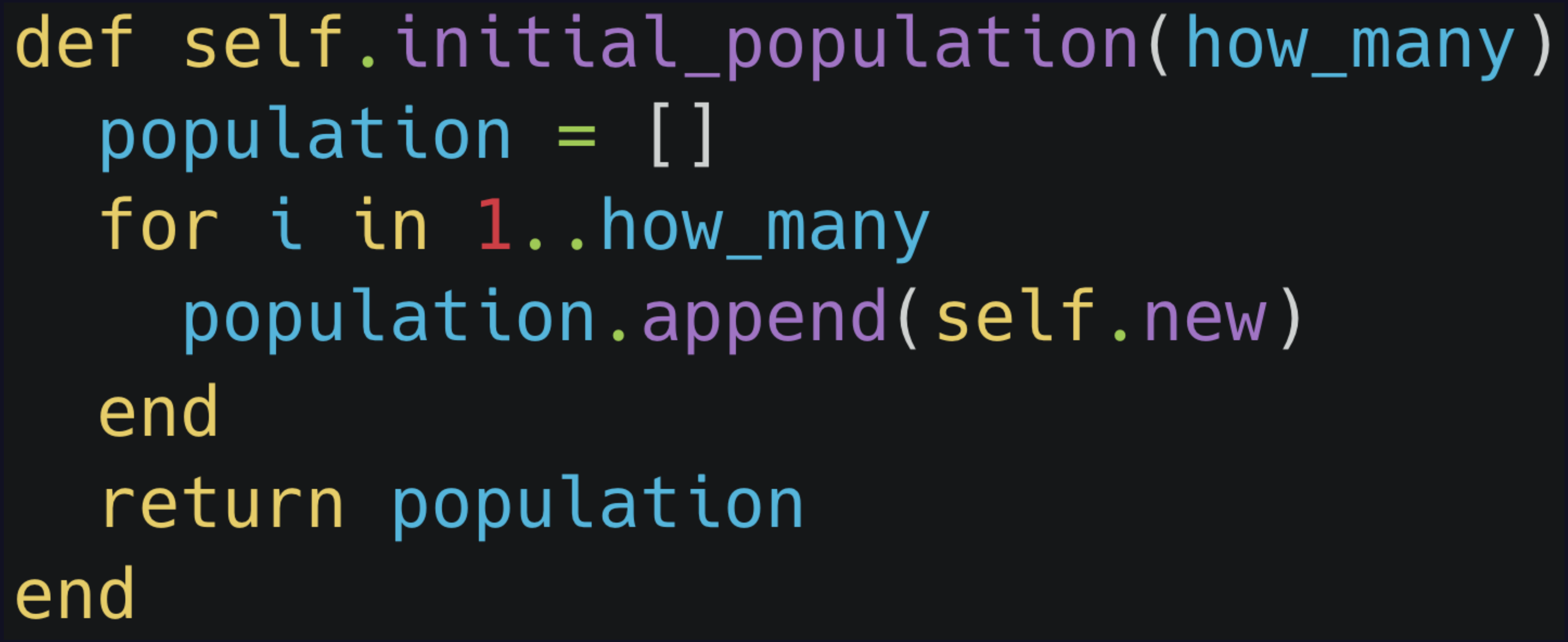


```
class Truckload
  def initialize()
    @contents = rand(128)
  end
end
```

Speaker notes

Whoa, that looks like we're just making a random number! That's right, we're making a random number with seven bits, so we have a random 1 or 0 for each of our seven possible items. We could get as complex as we want in this function, like dictating a minimum or maximum number of items, but let's keep it simple.

To create an initial population, we can just create a bunch of candidates and stuff them into an array, . . .



```
def self.initial_population(how_many)
  population = []
  for i in 1..how_many
    population.append(self.new)
  end
  return population
end
```

Speaker notes

. . . like so. (This could actually be done in much more idiomatic Ruby, so don't scold me for that, I'm just trying to keep it easily understandable by people who don't know Ruby.) So if we create a population of ten Truckloads, we might wind up with a list like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
Y	N	N	Y	N	Y	Y
N	N	N	Y	Y	N	N
N	Y	N	N	N	Y	N
N	Y	Y	N	Y	N	N
Y	Y	Y	N	Y	Y	N
Y	Y	N	Y	N	N	N
Y	N	N	Y	N	Y	N
Y	Y	N	N	N	N	N
N	N	Y	Y	Y	Y	Y
N	N	Y	N	N	Y	N

Why ten? Because that's what fits on the screen in a decently readable size. If I were doing this for real, with a much more complex domain, I might use a hundred, or a thousand, or even more.

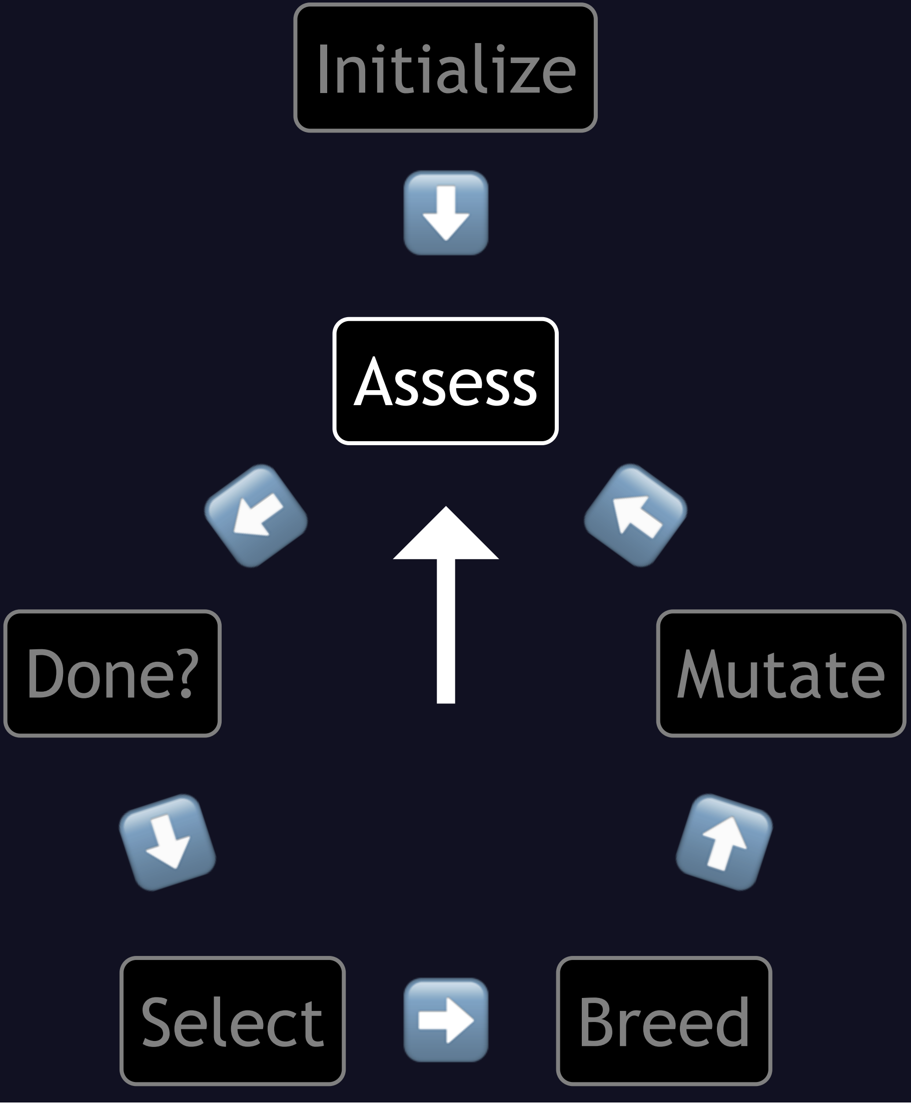
But how did we get from random numbers to those combinations? Behind the scenes, that translation might look like this:

```ruby
class Truckload
  class Item
    attr_reader :name, :weight, :value
    def initialize(name, weight, value)
      @name   = name
      @weight = weight
      @value  = value
    end
  end
  ITEMS = [
    #         name      weight  value
    Item.new("Cow",       1500,  2000),
    Item.new("Milk",      1720,   800),
    Item.new("Cheese",    1000, 12000),
    Item.new("Butter",    1000,  3000),
    Item.new("Ice Cream", 1000,  2000),
    Item.new("Meat",      1280,  8000),
    Item.new("Leather",   1100,  6000),
  ]
```

Speaker notes

We have a list of items we can take, as instances of an inner class describing them. To check what's in our cargo manifest, represented by our Truckload's contents value, we can iterate through the list of possible items, checking whether the corresponding bit is on.

Now that we're done with Initialization, we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . assess how "fit" each of these truckloads is. We do this with what's called a "fitness function". Just like how biological creatures might be perfectly fit for one environment but a lousy fit for another, this should reflect how fit a candidate is for some particular purpose. In this case, we already know we want the total value, BUT, any load that's too heavy for the truck, is worthless. In Ruby, that would look like this:

```ruby
def fitness()
  weight =
    (0 ... ITEMS.count).
    map { |n| bit_on?(n, contents) ? ITEMS[n].weight : 0 }.
    sum
  return 0 if weight > 4000
  (0 ... ITEMS.count).
    map { |n| bit_on?(n, contents) ? ITEMS[n].value : 0 }.
    sum
end
```

Speaker notes

We iterate through the possible items, summing up the weights of the ones we want to take. (Writing the bit_on? function is left as an exercise, to keep this code simple.) If that exceeds the truck's capacity, we return zero, else we use the same technique to sum up the values of those same items.

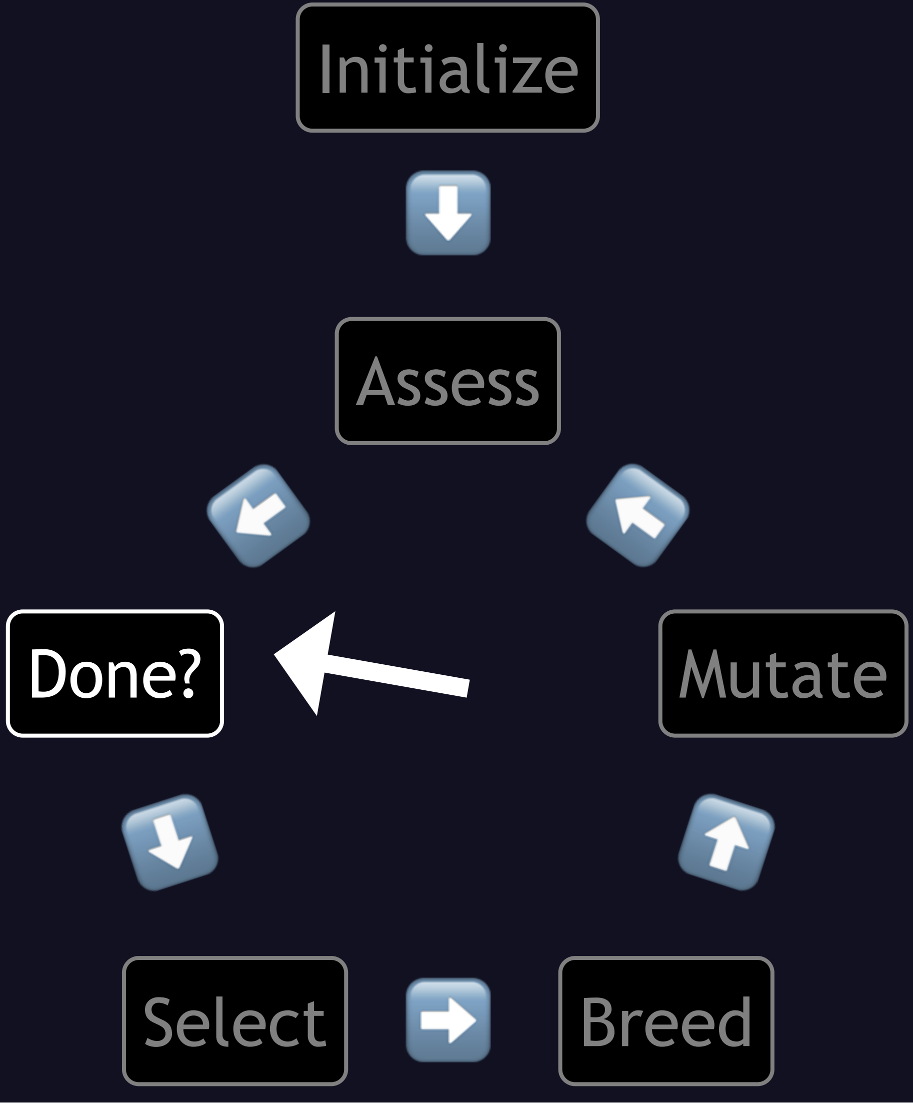
Again, we could get as complex as we want in this function, and NASA's antenna fitness function certainly must have been. For instance, if the truck were even smaller, and we had the volume of each item, we could also total up the volume, and make sure it all fits.

Anyway, if we run this fitness function on our population, we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
Y	N	N	Y	N	Y	Y	0
N	N	N	Y	Y	N	N	5,000
N	Y	N	N	N	Y	N	8,800
N	Y	Y	N	Y	N	N	14,800
Y	Y	Y	N	Y	Y	N	0
Y	Y	N	Y	N	N	N	0
Y	N	N	Y	N	Y	N	13,000
Y	Y	N	N	N	N	N	2,800
N	N	Y	Y	Y	Y	Y	0
N	N	Y	N	N	Y	N	20,000

Speaker notes

So now that we've assessed their fitness, we . . .



Initialize

Assess

Done?

Mutate

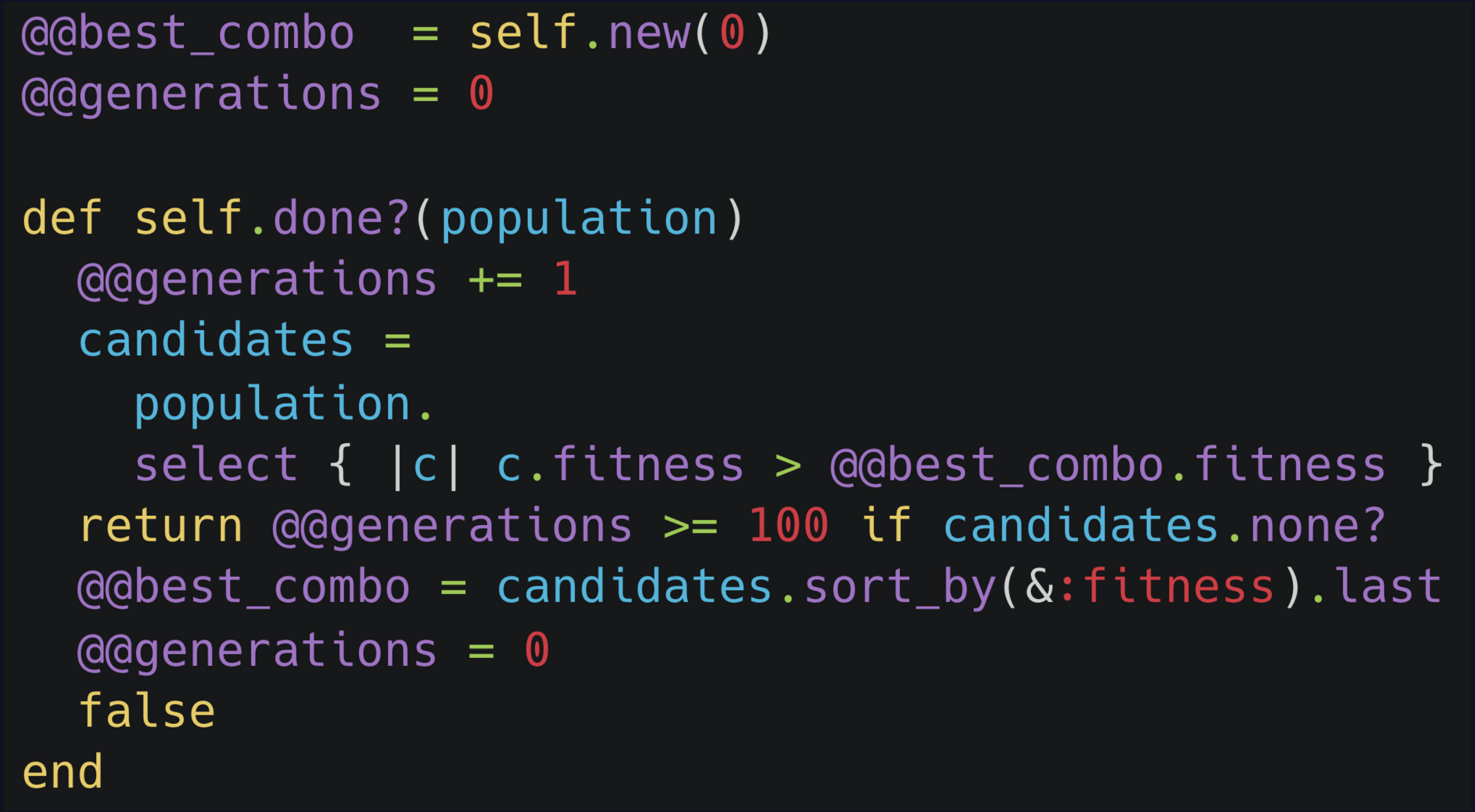
Select

Breed

Speaker notes

. . . check if we're done. So what are our criteria? The function can be simple, but it can take some thinking to figure out what the function should do. With a knapsack problem, a good solution can be made totally worthless by adding just one more item, and thereby exceeding the constraints. So, we're going to record the best we've seen, and stop if we haven't seen anything better within 100 generations.

Why 100? Pretty much random. It seems like enough for a good chance for improvement, and since what we're doing is so simple, and our population is so small, using lots of generations isn't very slow. In Ruby, that would look like this:



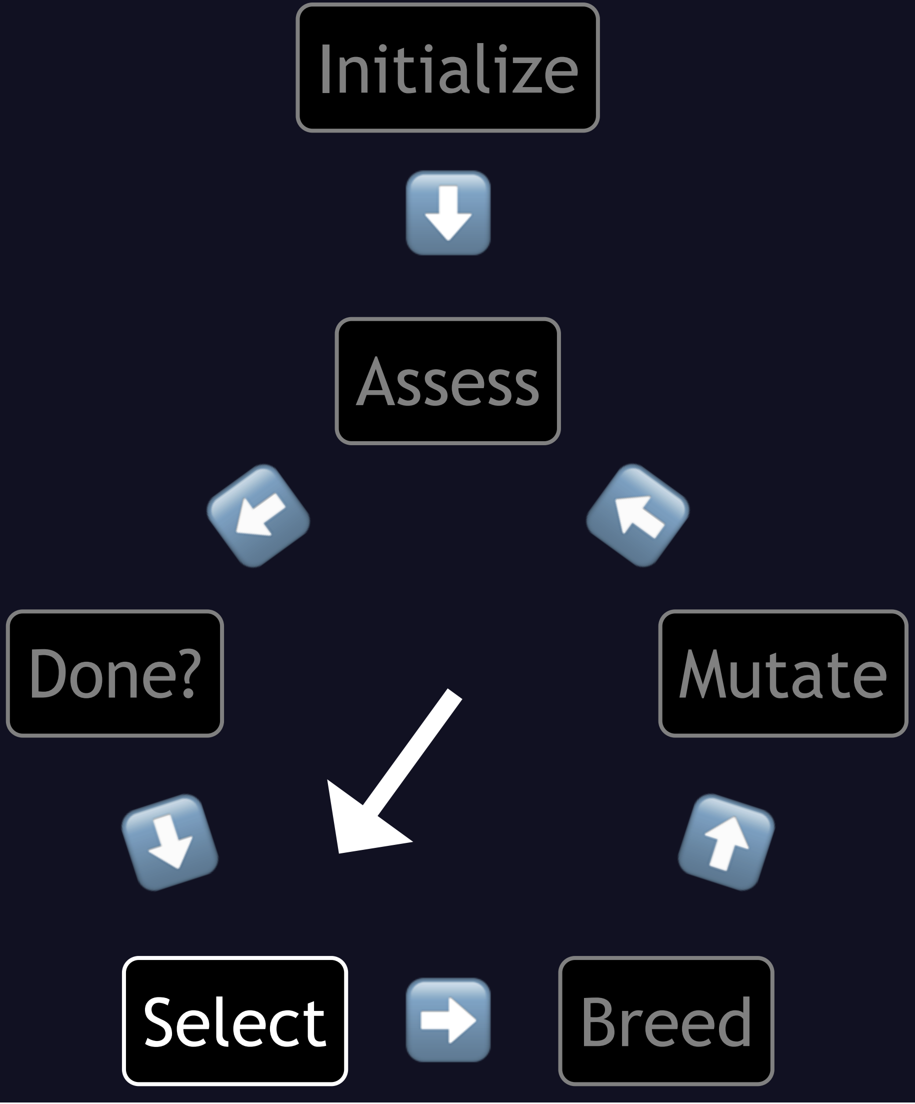
```ruby
@@best_combo  = self.new(0)
@@generations = 0

def self.done?(population)
  @@generations += 1
  candidates =
    population.
    select { |c| c.fitness > @@best_combo.fitness }
  return @@generations >= 100 if candidates.none?
  @@best_combo = candidates.sort_by(&:fitness).last
  @@generations = 0
  false
end
```

Speaker notes

When the code is initially parsed, we set the initial best combo as empty, and we set how many generations it's been since we saw that, as zero. When the function is called, we increment the number of generations, look at the fitness of each candidate in the current population, and select only the ones with a better fitness than the best one so far. If there are none, then we return true if it's been 100 generations since the best one, else we return false. If there is at least one candidate exceeding our benchmark, we sort them by fitness, take the fittest one, make that our new benchmark, and reset the generation counter.

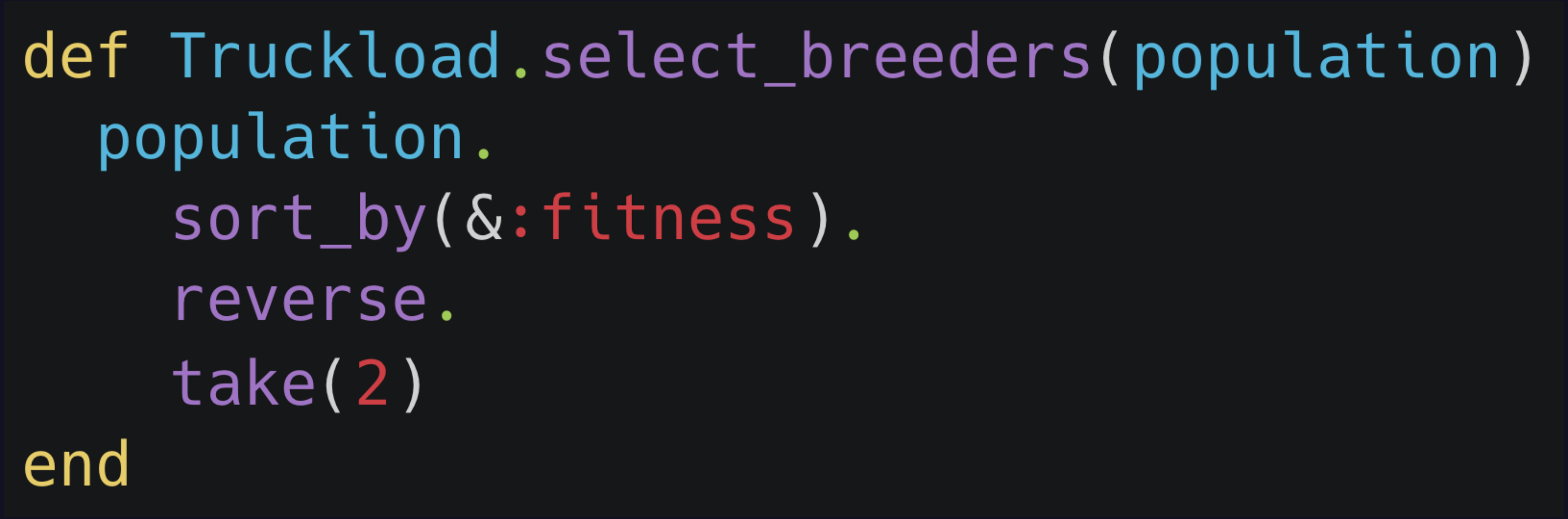
Again, we can get as complex as we want, not only in checking the maximum fitness, but we could look at other stopping criteria, like the average or minimum fitness, or achieving some specific level of maximum fitness, like some maximum number of generations, or amount of time, (whether clock time or CPU or whatever), or a STOP button, or many other ways, or a combination of ways.

Since we're not done, the next step is to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . select some candidates to breed the next generation. The obvious way is to take the top two most fit, like so:



```ruby
def Truckload.select_breeders(population)
  population.
    sort_by(&:fitness).
    reverse.
    take(2)
end
```

Speaker notes

We take the population, sort them by fitness in descending order, and take the first two. Out of our current population, we would choose:

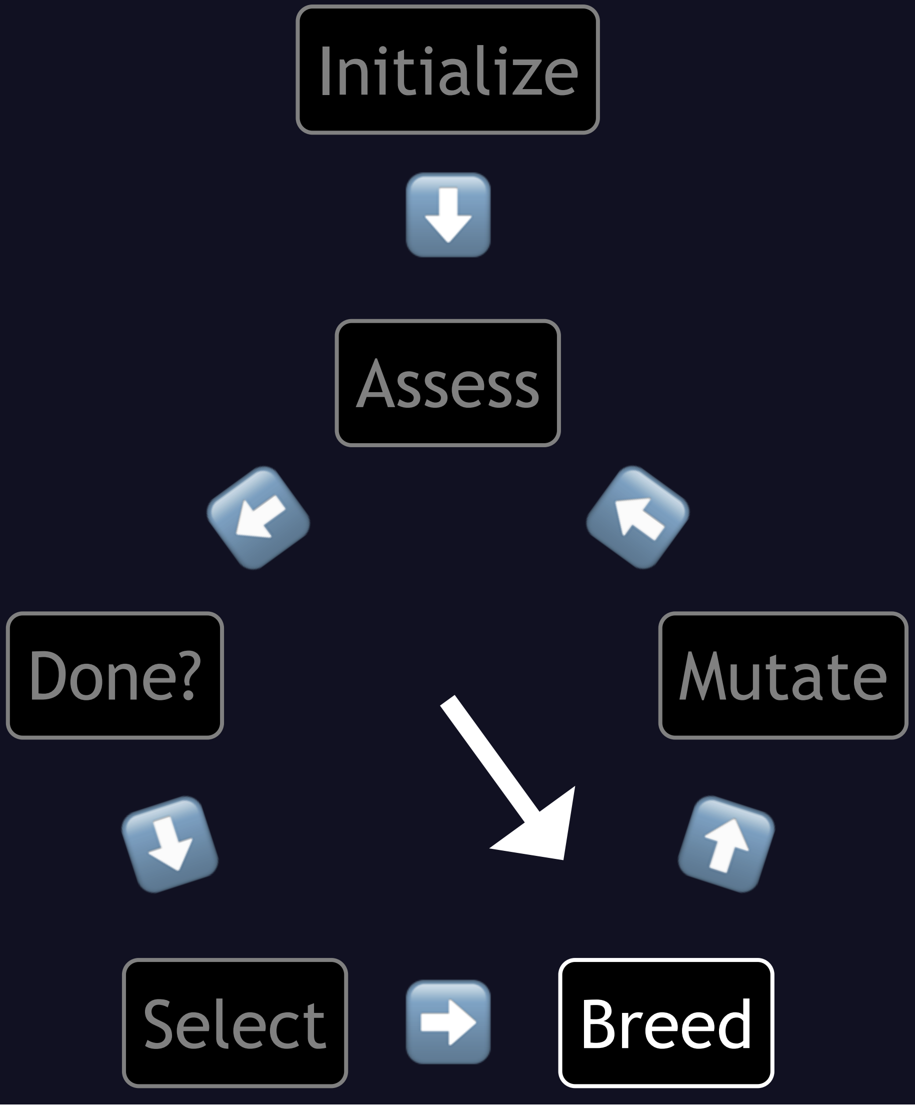
Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800

Speaker notes

these two. As usual, we also could get more complicated, like selecting two randomly with each candidate having a chance to be selected, proportional to their fitness, called "Roulette Wheel" selection, and lots of other ways.

We could also take more than two, whether to breed all pairs in that set or to combine more than two at once. Or we could combine strategies, such as using all trios from a randomly chosen five, with the fitter ones having a better chance to be chosen.

Now that we've chosen our breeders, next we . . .



Initialize

Assess

Done?

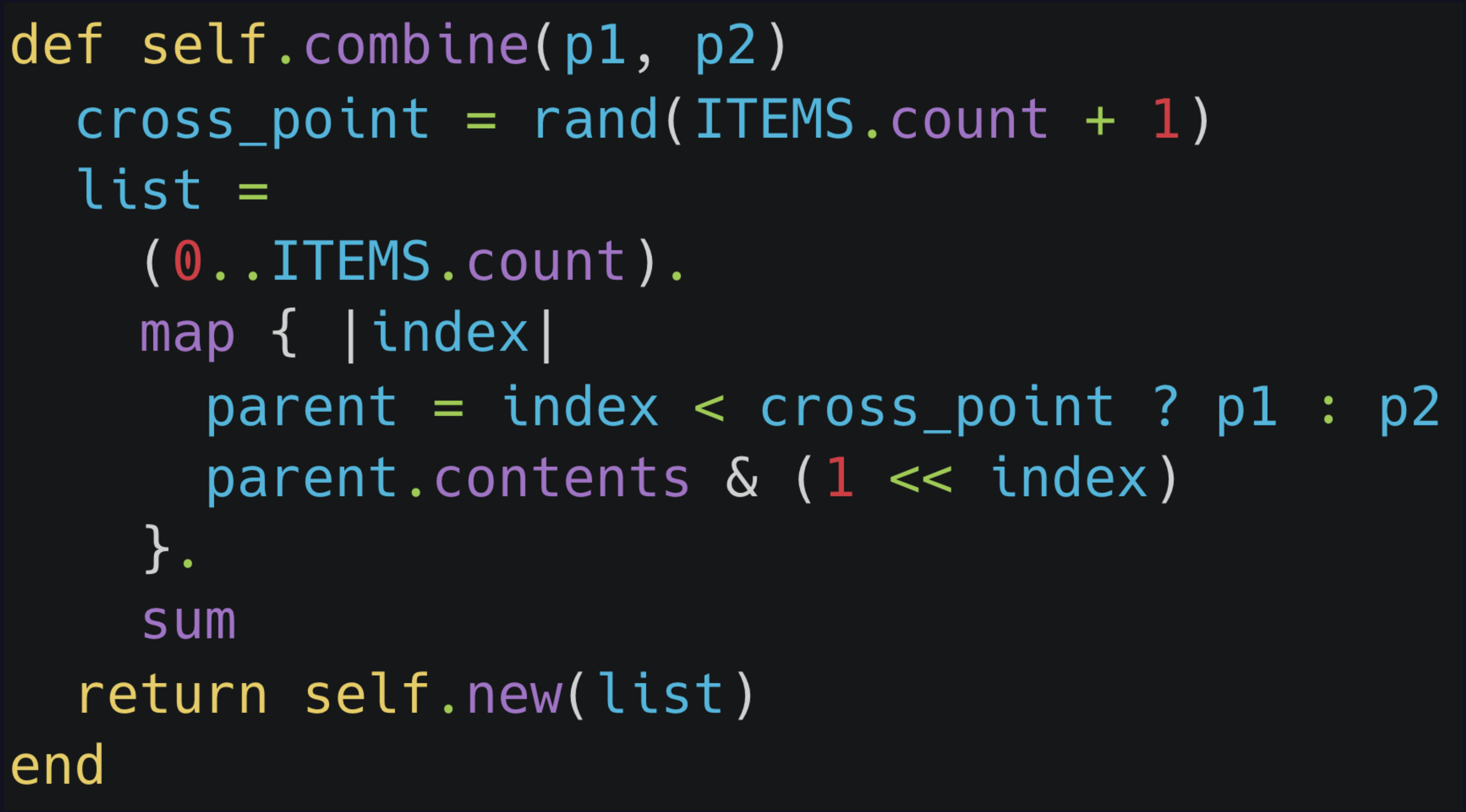
Mutate

Select

Breed

Speaker notes

. . . breed them. The usual way is called crossover. This consists of taking the data points, or in Genetic Algorithm lingo, the "genes", from one parent, up to some randomly chosen crossover point, then switching to the other parent. This can be extended with multiple crossover points, but we're just going to use one, like so:



```ruby
def self.combine(p1, p2)
  cross_point = rand(ITEMS.count + 1)
  list =
    (0..ITEMS.count).
    map { |index|
      parent = index < cross_point ? p1 : p2
      parent.contents & (1 << index)
    }.
    sum
  return self.new(list)
end
```

Speaker notes

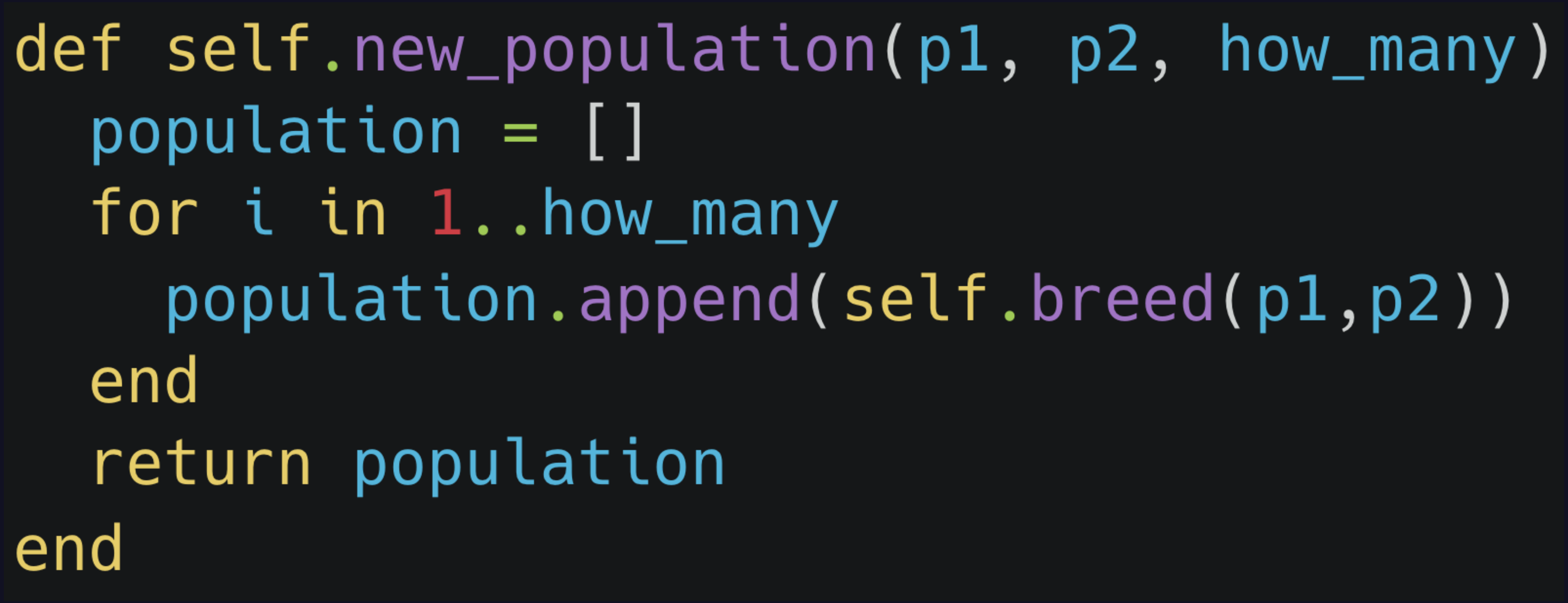
We establish the crossover point for each new candidate, as a random number between zero and how many items there are, inclusive. Then we iterate through the list of items, by index number. If we haven't yet hit the crossover point, we get the decision for that item from the first parent, else we get it from the other parent. This means that it could be all copied from one parent or the other, or it could switch at some point. If we do this once, with a crossover point of 3, so we take 3 values from the first parent, that would get us a result like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	N	Y	N
			+			
N	Y	Y	N	Y	N	N
			=			
N	N	Y	N	Y	N	N

Cow Milk Cheese Butter Ice Cream Meat Leather

Speaker notes

But this is just one of ten results, because we're making a whole new population, like so:



```
def self.new_population(p1, p2, how_many)
  population = []
  for i in 1..how_many
    population.append(self.breed(p1,p2))
  end
  return population
end
```

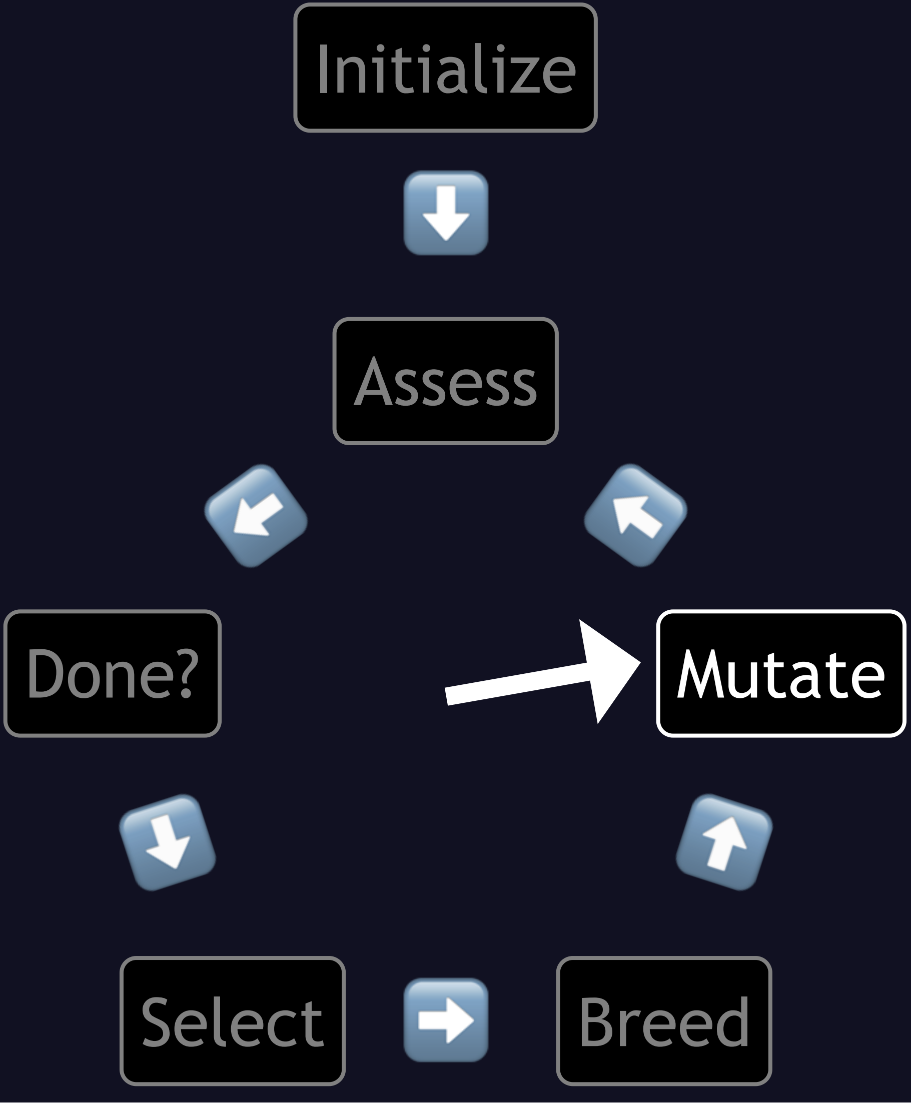
Speaker notes

This is just like how we created the initial population, except that instead of each candidate being made from scratch, they're the product of breeding our chosen breeders. The whole list might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	Y	N	N
N	Y	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N

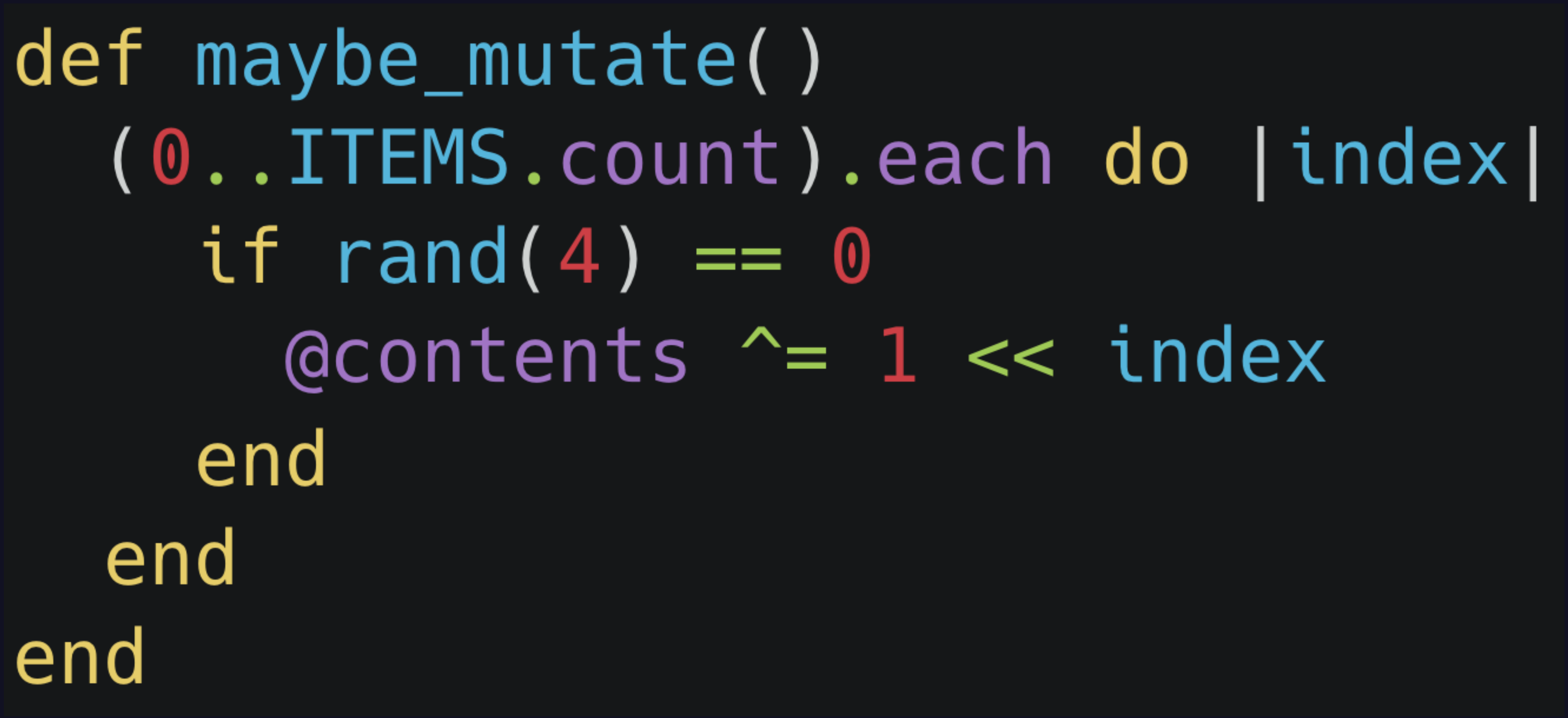
Speaker notes

Lots of family resemblance there, eh? None of these loads include a cow, butter, or leather, and they all include cheese. That's because both of our two breeders were like that. If we were to just continue breeding the fittest of each generation, we wouldn't ever see any loads including a cow, butter, leather, or no cheese, but we fix that in the next step, which is to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . mutate them. Again, I'm going to keep it very simple, and give each gene a 1 in 4 chance of flipping. In code, that looks like this:



```ruby
def maybe_mutate()
  (0..ITEMS.count).each do |index|
    if rand(4) == 0
      @contents ^= 1 << index
    end
  end
end
```

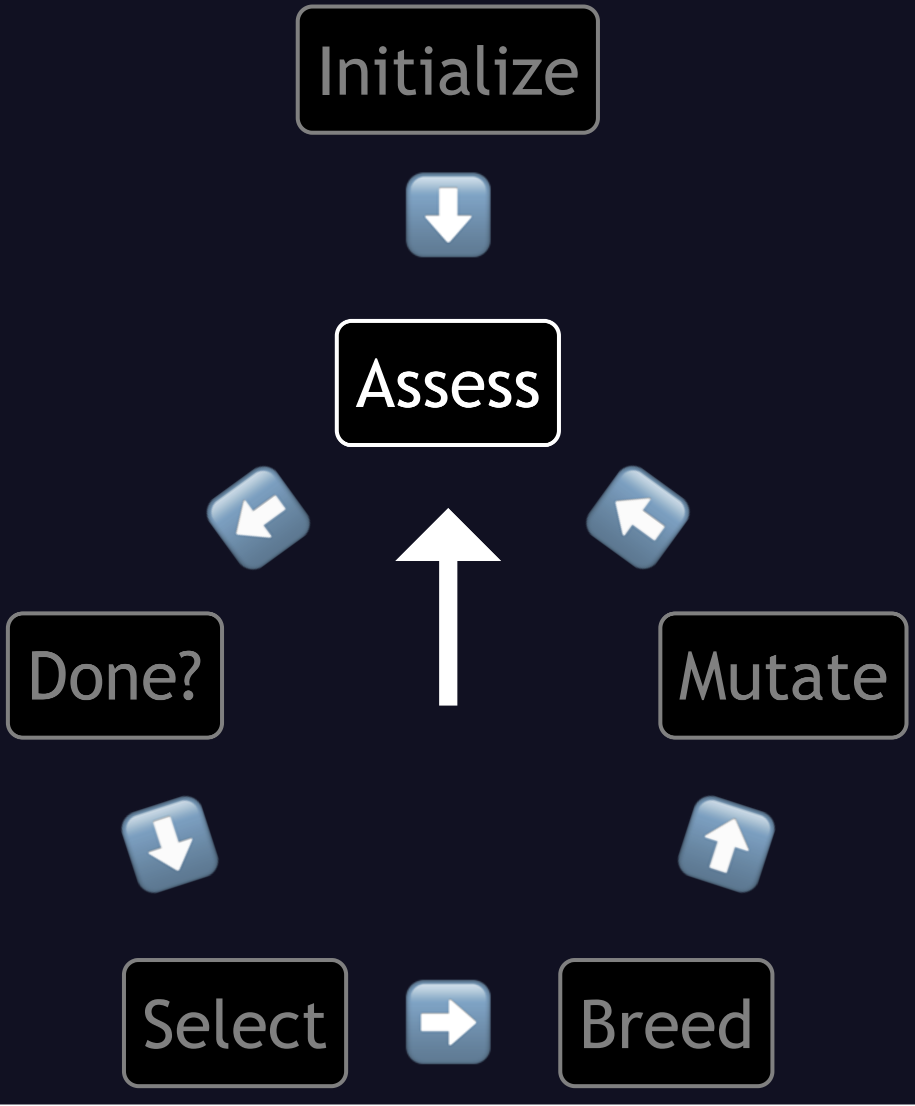
Speaker notes

We iterate through the item numbers, and for each one, if a random number from zero to three is a zero, we flip that bit. Again, we could get as complex as we want, like having some genes more likely to mutate than others, or having some minimum or maximum number of mutations per candidate, or all kinds of other options. If we run this mutation function on these new candidates, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	Y	N	Y	N	N	Y
N	N	Y	Y	N	N	N
Y	N	Y	Y	Y	Y	N
Y	Y	Y	Y	Y	Y	N
Y	Y	Y	Y	N	N	Y
N	Y	N	Y	N	N	Y
Y	Y	N	Y	N	N	Y
N	N	Y	N	Y	N	N
Y	N	N	Y	Y	N	N
N	N	Y	N	Y	N	N

Speaker notes

. . . where green means that it changed. You can see that we now DO have some truckloads that include a cow, butter, or leather, or no cheese. Now we go back to . . .



Initialize

Assess

Done?

Mutate

Select

Breed

Speaker notes

. . . assessing the fitness of these new candidates. If we sort on fitness descending, just to make it easy to find the best, we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	Y	N	N	N	15,000
N	N	Y	N	Y	N	N	14,000
N	N	Y	N	Y	N	N	14,000
N	Y	N	Y	N	N	Y	9,800
N	Y	N	Y	N	N	Y	9,800
Y	N	N	Y	Y	N	N	7,000
Y	Y	Y	Y	N	N	Y	0
Y	Y	Y	Y	Y	Y	N	0
Y	N	Y	Y	Y	Y	N	0
Y	Y	N	Y	N	N	Y	0

Speaker notes

Oh noes! Our maximum fitness actually went down! As you may recall, our previous best one scored 20,000. But don't worry, as you may recall from our "are we done yet" function, we hang onto the best one, and just try to outdo it, so we haven't lost it. The next generation might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	Y	Y	N	N	Y	N	20,800
N	N	Y	N	N	Y	N	20,000
N	N	Y	N	N	Y	N	20,000
N	N	N	N	N	Y	Y	14,000
N	N	Y	N	N	N	N	12,000
N	Y	N	N	Y	N	N	2,800
Y	N	Y	Y	N	Y	N	0
Y	Y	Y	N	Y	Y	N	0
N	Y	Y	N	Y	Y	N	0
N	Y	Y	Y	N	Y	N	0

Speaker notes

. . . a small improvement over our prior best! So, we set that top one as our benchmark, and reset the counter of generations since we saw it. If we let this run to completion, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	Y	26,000
N	N	Y	Y	N	N	Y	21,000
N	Y	N	N	Y	Y	N	10,800
Y	N	N	N	N	N	Y	8,000
N	N	N	N	N	Y	N	8,000
N	N	N	Y	Y	N	N	5,000
N	Y	N	N	N	N	N	8,00
N	Y	N	N	N	N	N	8,00
Y	Y	N	Y	N	N	Y	0
Y	Y	N	Y	Y	Y	Y	0

Speaker notes

. . . with our best truckload scoring 26,000, made up of cheese, meat, and leather. So that's one complete run of a genetic algorithm. If we wanted to check whether that was the best that this algorithm could produce, we could just run it again, as many times as we like, within reason, since it's so much faster than brute force. Okay, maybe writing all this code is not so much faster when we've only got seven items, and such simple criteria, but if we had to choose among many more items, with more complex criteria, for many truckloads a day, this might be more worthwhile.

Now, suppose we want to evolve . . .

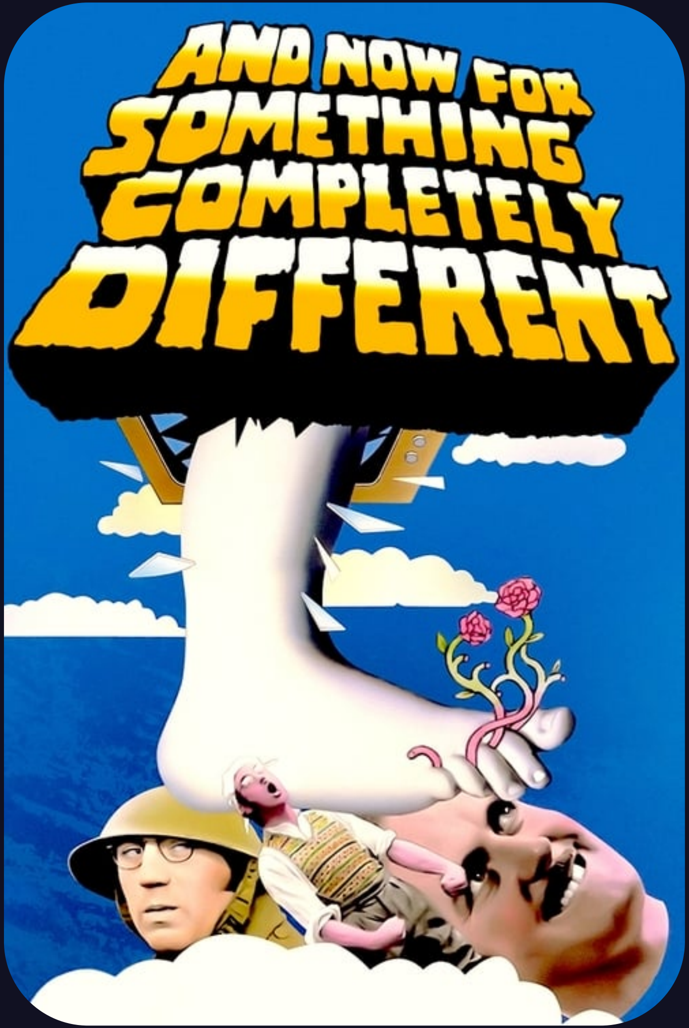


Image: https://image.tmdb.org/t/p/w500/ajbdFQLvJTlNu4LnVWGnNMb4mZ8.jpg (used for edu fair use)

Speaker notes

. . . something completely different. Suppose we want to "evolve" a good set of stats for a Dungeons and Dragons fighter character, so our candidates are tuples of numbers, rather than strings of bits. D&D character stats are . . .

STRength

INTelligence

DEXterity

CONstitution

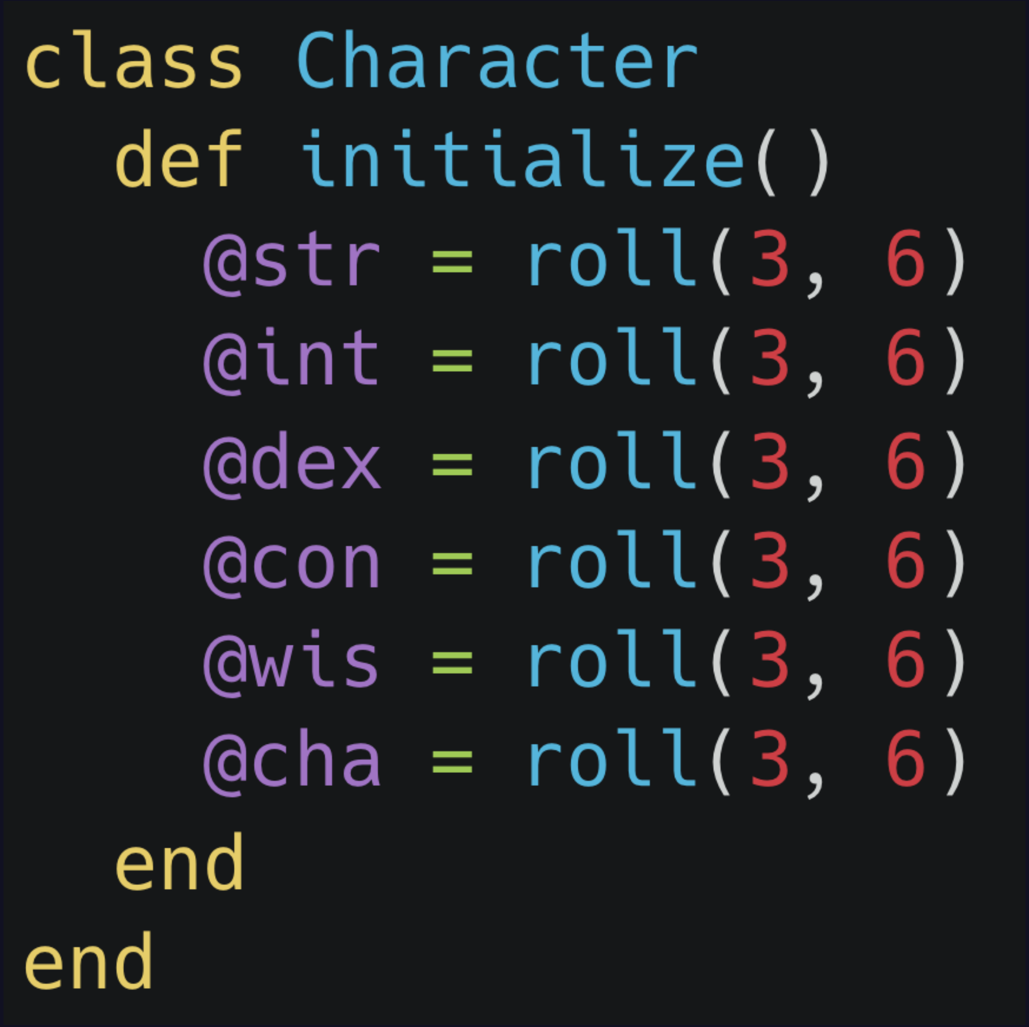
WISdom

CHArisma

3d6 each

ignoring STR 18/xx

. . . Strength, Intelligence, Dexterity, Constitution, Wisdom, and Charisma, each determined by rolling three six-sided dice, or 3d6 for short. (I'm going to gloss over how you can sometimes have extra strength.) In Ruby that might look like this:



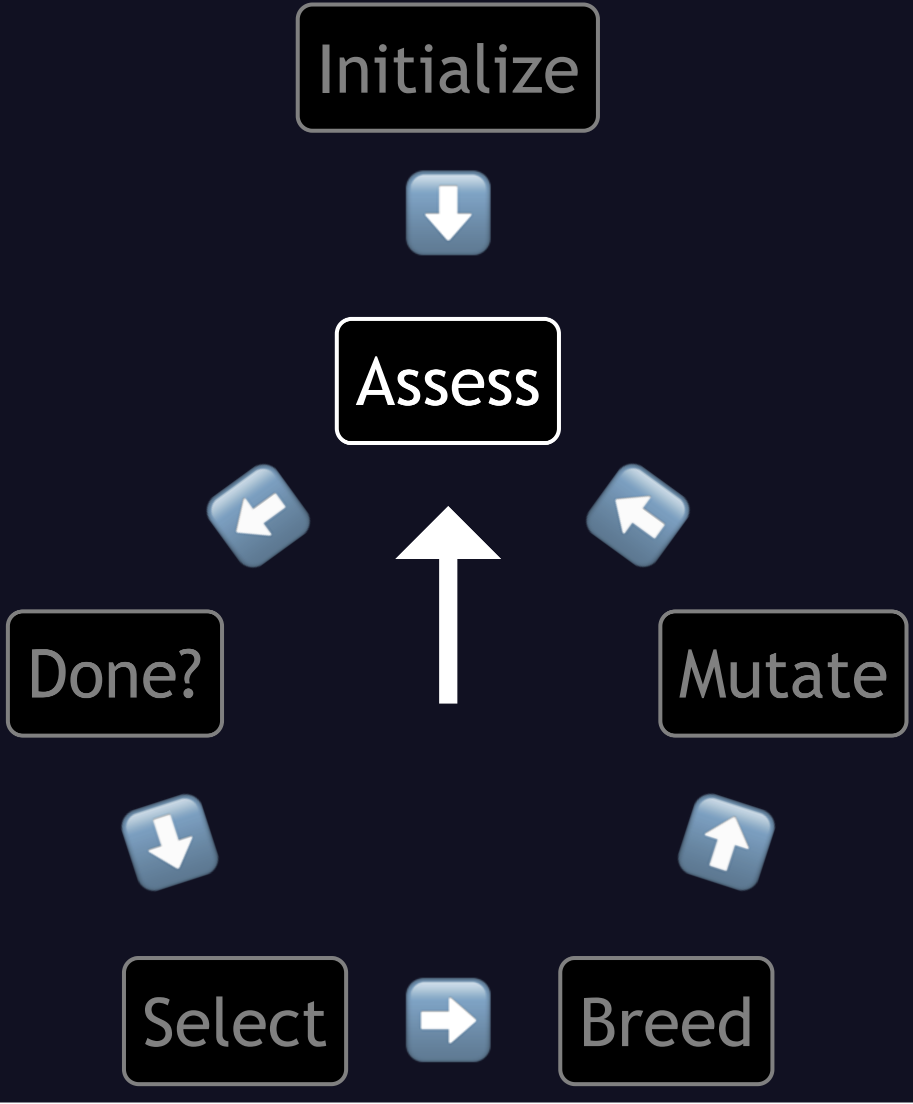
```ruby
class Character
  def initialize()
    @str = roll(3, 6)
    @int = roll(3, 6)
    @dex = roll(3, 6)
    @con = roll(3, 6)
    @wis = roll(3, 6)
    @cha = roll(3, 6)
  end
end
```

Speaker notes

(Defining the roll function is left as an exercise.) So if we create an initial population of ten Characters, it might look like this:

Str	Int	Dex	Con	Wis	Cha
11	9	9	10	7	15
4	14	8	12	13	10
9	14	15	11	9	16
14	15	10	7	6	14
13	12	7	13	11	10
12	12	10	9	5	16
11	12	9	13	6	12
10	14	12	8	8	16
14	7	8	9	8	8
14	12	13	5	13	13

So that's the Initialize step. The next step is . . .



Initialize

Assess

Done?

Mutate

Select

Breed

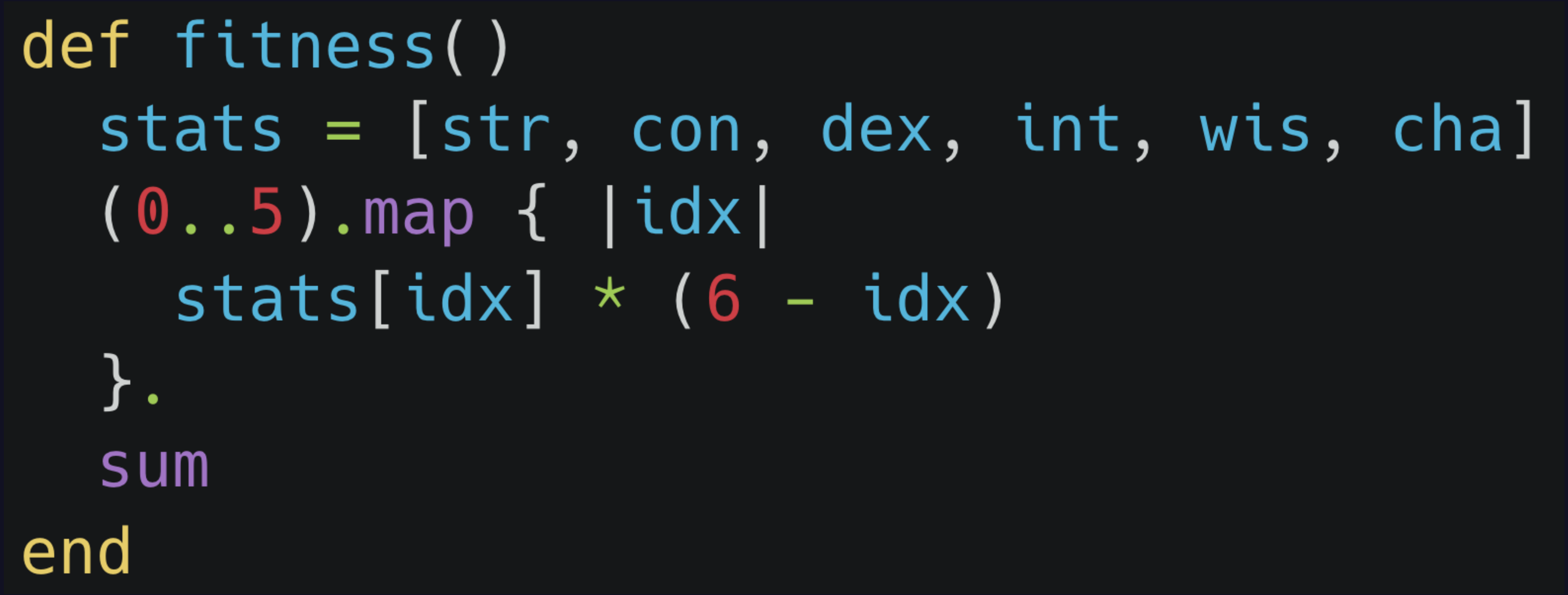
Speaker notes

. . . to assess how "fit" each one of these characters is. We're trying to evolve a good set of Fighter stats, so it should be based mainly on strength and constitution. Dexterity is also helpful. Intelligence, wisdom, and charisma, not so much, but we don't want them too low, for the sake of occasional saving throws. I tried several different things, such as . . .

```
def fitness()
  str * 2 + con + dex / 2
end
```

Speaker notes

totaling up double the strength, the constitution, and half the dexterity. But, the other stats tended to get too low, and even the dexterity. So I tried . . .



```
def fitness()
  stats = [str, con, dex, int, wis, cha]
  (0..5).map { |idx|
    stats[idx] * (6 - idx)
  }.
  sum
end
```

Speaker notes

prioritizing them linearly, adding up six times the strength, five times the constitution, and so on down to one times the charisma. But then the other stats got too high, and the characters seemed too generalized. So I finally settled on this:

```
def fitness()
  stats = [str, con, dex, int, wis, cha]
  (0..5).map { |idx|
    stats[idx] * 2 ** (5 - idx)
  }.
  sum
end
```

Speaker notes

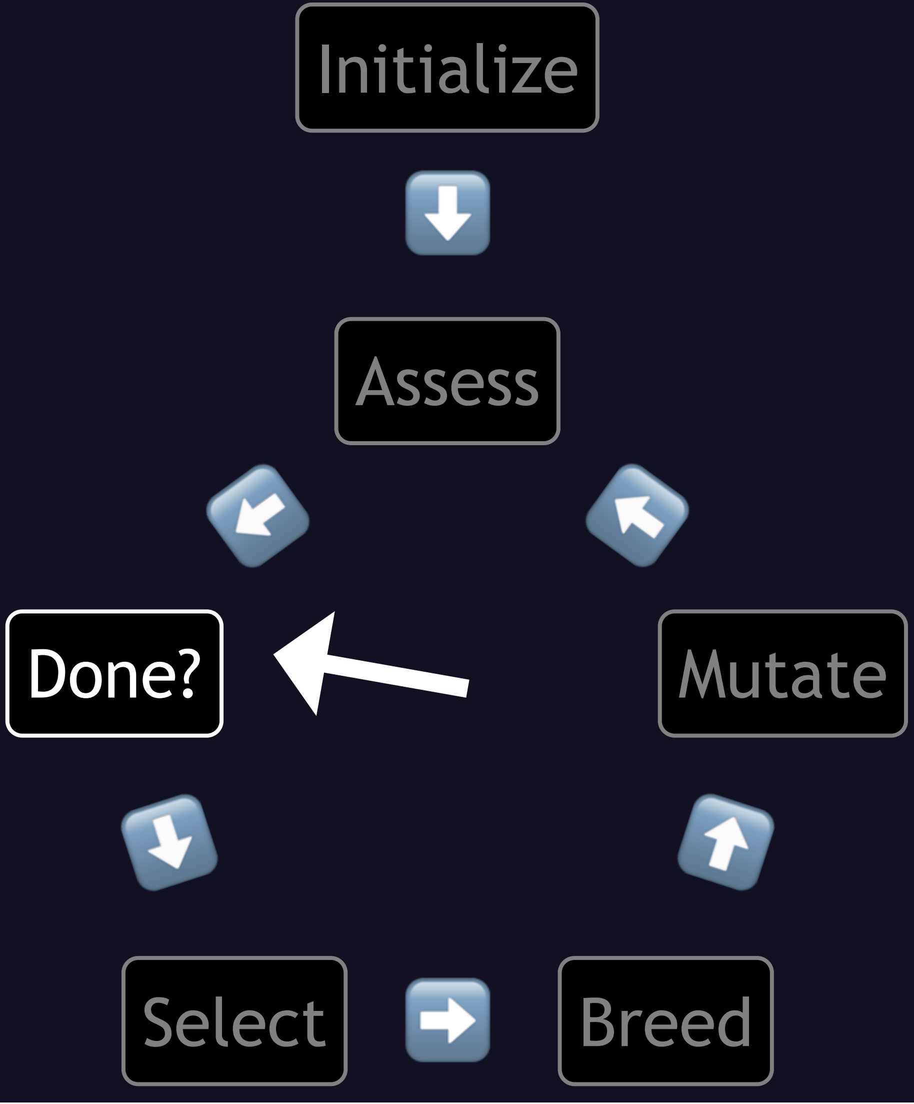
. . . prioritizing the stats again but much more strongly, totaling up 32 times the strength, 16 times the constitution, and so on down to one times the charisma. Here we see that even though the fitness function itself can be very simple, it can be difficult to figure out one that will yield good results.

If we run this on our population, we get this:

Str	Int	Dex	Con	Wis	Cha	Fit
11	9	9	10	7	15	649
4	14	8	12	13	10	476
9	14	15	11	9	16	674
14	15	10	7	6	14	726
13	12	7	13	11	10	760
12	12	10	9	5	16	682
11	12	9	13	6	12	703
10	14	12	8	8	16	632
14	7	8	9	8	8	708
14	12	13	5	13	13	719

Speaker notes

Now that we've assessed their fitness, we can ask, are we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

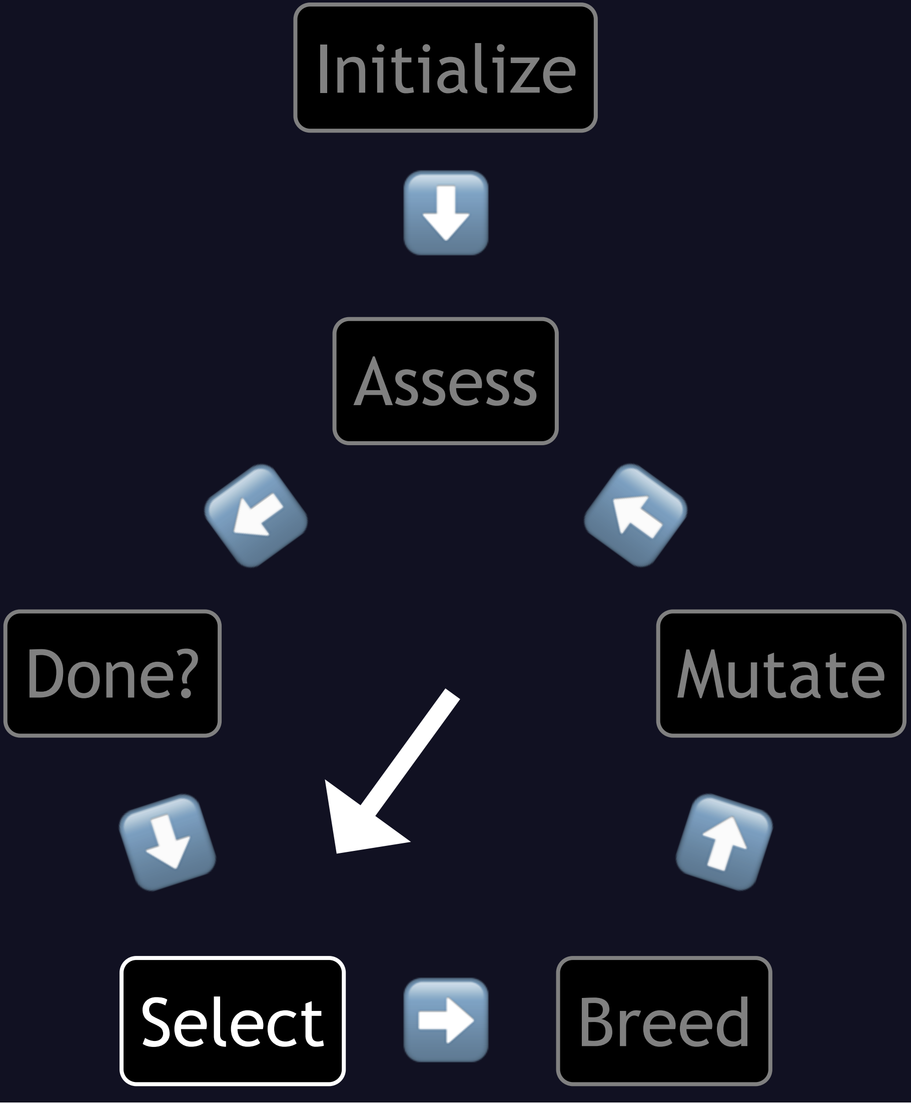
Speaker notes

. . . done? What are our criteria? Let's say we're done if any candidates get 90% of the way to the maximum score of our fitness function. I'll spare you the math, but that would be 1,021. In code, checking that would look like this:

```ruby
def Character.done?(population)
  population.any? { |cand|
    cand.fitness >= 1021
  }
end
```

Speaker notes

Very simple. None of our current candidates score anywhere near 1021, so we . . .

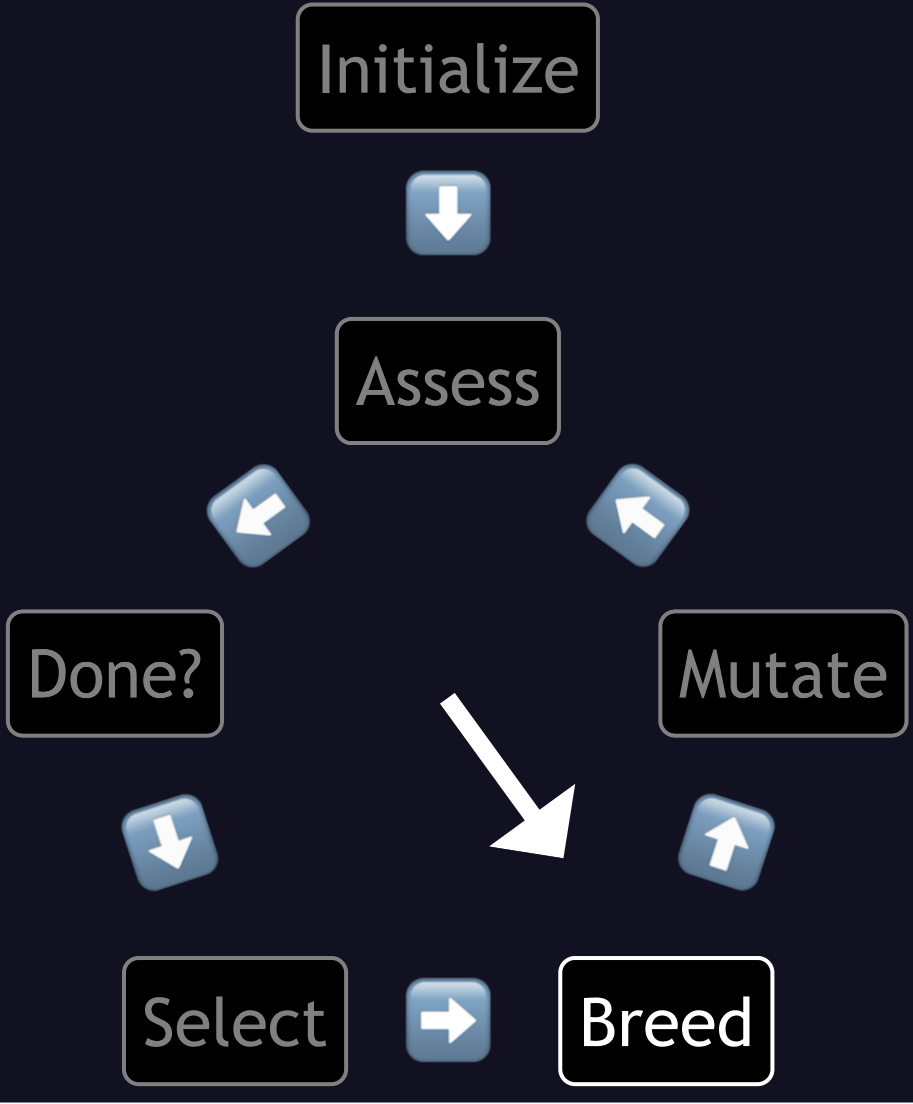


Initialize
Assess
Done?
Mutate
Select
Breed

. . . select some candidates to breed the next generation. Taking the top two scorers again we get:

Str	Int	Dex	Con	Wis	Cha	Fit
13	12	7	13	11	10	760
14	15	10	7	6	14	726

these two. The abstraction gets a bit more obviously leaky now, because we're ignoring sexes; we have no guarantee that these characters will be a male and a female, as we're not even making that part of the data. We had the same leak last time, with the Truckloads, but then it was not so relevant, so I let it slide. Now that they're living beings (whether humans or elves or whatever), though, we could add such complications, and many other such factors. That would complicate our breeder selection enormously, maybe even make it impossible to find a viable pair in such a small population. Anyway, next we actually . . .



Initialize

Assess

Done?

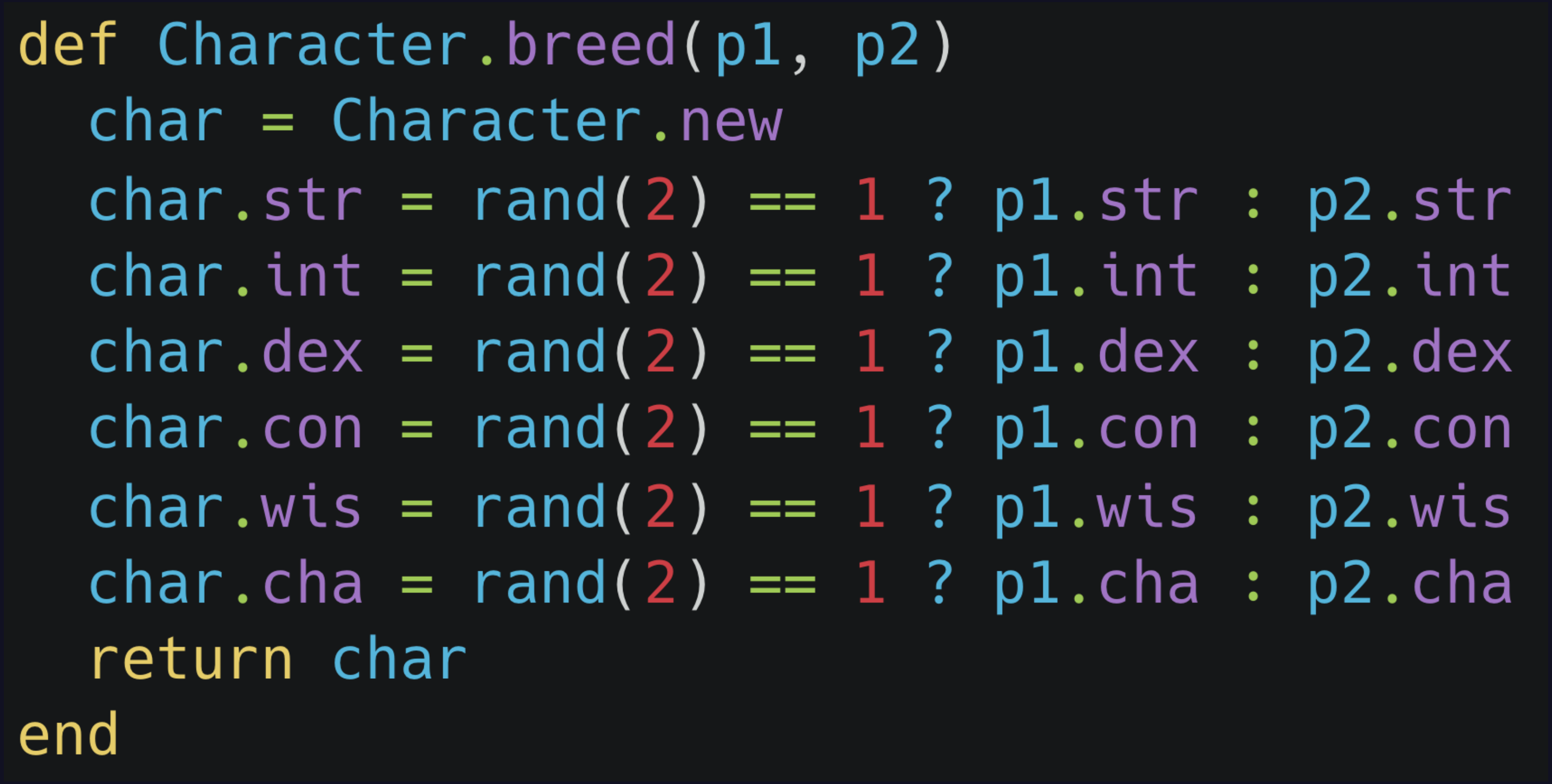
Mutate

Select

Breed

Speaker notes

. . . breed our chosen pair, this time using another common strategy, of essentially flipping a coin for each gene, like so:



```ruby
def Character.breed(p1, p2)
  char = Character.new
  char.str = rand(2) == 1 ? p1.str : p2.str
  char.int = rand(2) == 1 ? p1.int : p2.int
  char.dex = rand(2) == 1 ? p1.dex : p2.dex
  char.con = rand(2) == 1 ? p1.con : p2.con
  char.wis = rand(2) == 1 ? p1.wis : p2.wis
  char.cha = rand(2) == 1 ? p1.cha : p2.cha
  return char
end
```

Speaker notes

We go through the stats one by one, flip a coin (or "roll a d2"), and if it comes up 1, we get that stat from the first parent, else we get it from the other parent. That could get us a result like this:

Str	Int	Dex	Con	Wis	Cha
13	12	7	13	11	10
			+		
14	15	10	7	6	14
			=		
13	15	10	13	6	10

Speaker notes

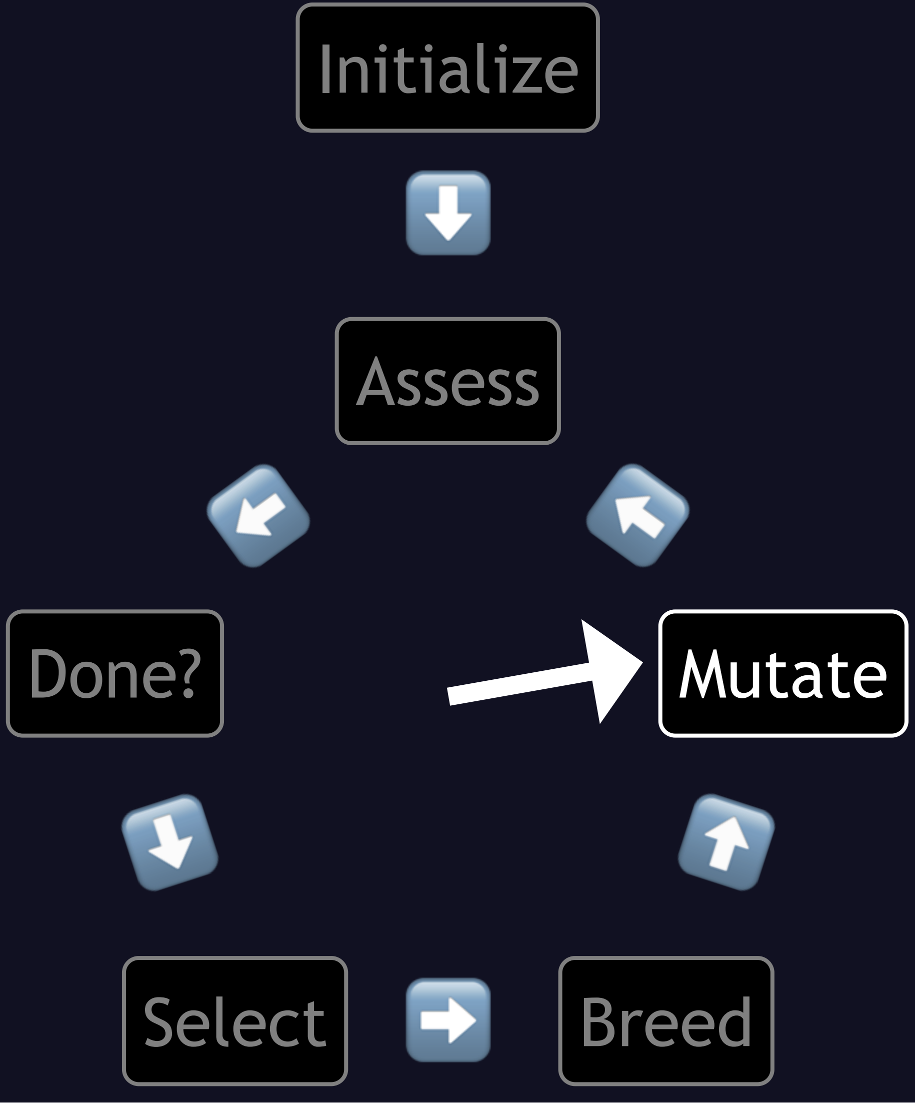
I've colored the ones from the first parent in green, and the second in red, to show the mixing. But again, this is just one of ten results, because we're making a whole new population, which might look something like this:

Str	Int	Dex	Con	Wis	Cha
13	12	10	7	6	14
13	12	7	13	6	14
14	12	10	13	11	14
14	15	7	13	6	14
13	12	10	13	6	14
14	15	10	7	11	10
14	12	10	13	6	10
13	15	10	13	6	14
13	15	10	13	6	10
14	12	7	7	6	14

Speaker notes

There are two things to notice here. First, the number of red and green is not always the same, neither in a single candidate nor the whole population. It's a series of random coin flips, so on average there will be three and three, but it could be anything up to six and zero either way. Second, notice the family resemblance! For each stat, there are only two possible values, or in Genetic Algorithm terms, "alleles", for a total of 64 possible combinations. There would be only one possible value, and therefore half as many possible combinations, if any stats were the same between the parents.

At a glance, these look on average much more suitable as fighters than the previous generation. (We'll figure their actual fitness scores later.) If we were to just continue breeding the fittest of each generation, we wouldn't see any change, let alone improvement, in the possible values of each stat, in other words, the alleles for each gene. For instance, the Wisdom would never be anything other than 6 or 11. But again, we fix that in the next step, which is to . . .



Initialize

Assess

Done?

Mutate

Select

Breed

Speaker notes

. . . mutate them. Again, I'm going to keep it very simple, and give each stat a 1/3 chance of staying the same, going up a point, or going down a point, within the valid range. In code, that looks like this:

```
def maybe_mutate()
  @str = maybe_mutate_stat(@str)
  @int = maybe_mutate_stat(@int)
  @dex = maybe_mutate_stat(@dex)
  @con = maybe_mutate_stat(@con)
  @wis = maybe_mutate_stat(@wis)
  @cha = maybe_mutate_stat(@cha)
end

def maybe_mutate_stat(stat)
  (stat + rand(3) - 1).clamp(3, 18)
end
```

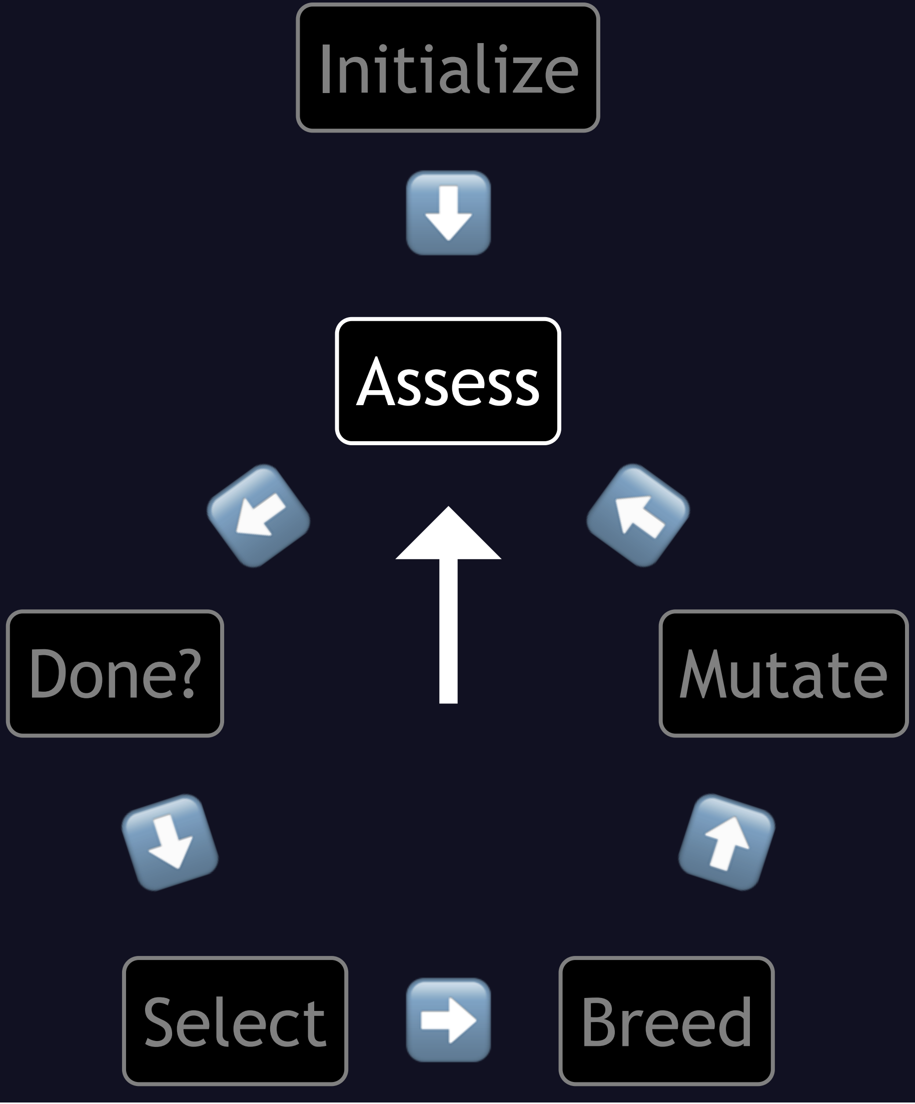
Speaker notes

For each stat, we add a random number from 0 to 2, and subtract one, which is like adding a random number from -1 to 1, but we clamp it to the range of 3 to 18. Again, we could get as complex as we want, like giving it a higher chance of going up or down, maybe by multiple points, if it's very low or very high, to simulate the real-world phenomenon of regression to the mean, or many other options. If we run this on our new population, we wind up with something like this:

Str	Int	Dex	Con	Wis	Cha
14	12	10	7	7	13
12	13	7	14	6	14
13	12	10	14	11	15
15	16	7	14	6	15
13	11	10	12	5	15
13	15	11	7	11	10
14	13	9	12	6	9
14	15	9	13	6	15
13	15	11	12	5	9
13	12	6	6	7	13

Speaker notes

. . . where green means it went up, and red means down. Looking at the values in each column, you can see it's now much more diverse. Now we go back to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . Step 2, and assess the fitness of these new candidates. If we sort on fitness descending, just to make it easy to find the best, we get this:

Str	Int	Dex	Con	Wis	Cha	Fit
15	16	7	14	6	15	851
14	15	9	13	6	15	815
13	12	10	14	11	15	805
14	13	9	12	6	9	785
13	15	11	12	5	9	775
13	11	10	12	5	15	757
12	13	7	14	6	14	742
14	12	10	7	7	13	715
13	15	11	7	11	10	708
13	12	6	6	7	13	635

Speaker notes

We can see that this generation is much improved from the prior one. The old one ranged from 476 to 760, and the new one from 635 to 851. It's still nowhere near our stopping criterion of 1021, so fast-forwarding through six more rounds, we finally get . . .

Str	Int	Dex	Con	Wis	Cha	Fit
18	18	9	18	4	13	1029
18	17	7	18	6	14	1014
18	16	8	18	3	13	1011
18	15	7	18	3	13	999
18	16	8	17	4	13	997
18	16	6	18	3	15	997
17	18	8	18	6	13	993
17	16	9	18	3	14	988
18	16	6	17	3	13	979
17	16	8	17	4	12	964

Speaker notes

. . . one suitable character, with 18 Strength and Constitution, acceptable though sub-par Dexterity, and surprisingly high Intelligence. That's not a problem, just a bit of a waste. If we really wanted it more specialized, we could complicate the fitness function further, and do things like explicitly demand well above average scores in the class's useful stats, and forbid it to be so high in the others, or at least apply a penalty.

So we've evolved a set of Fighter stats. Let's suppose we don't need any more Fighters in the party . . . but now we need a Wizard. All we need to do is tweak our fitness function, like so:

```ruby
def fitness()
  # below is the only line that changed!
  stats = [int, wis, dex, con, cha, str]
  (0..5).
    map { |idx| stats[idx] * 2 ** (5 - idx) }.
    sum
end
```

Speaker notes

. . . to prioritize intelligence first, then wisdom, dexterity, and so on, down to strength. An initial population would look roughly the same, since we haven't changed how that is generated, so I'll spare you those steps, but after 11 generations I got . . .

Str	Int	Dex	Con	Wis	Cha	Fit
12	18	17	12	15	18	1048
14	18	16	9	15	16	1026
15	18	15	10	15	16	1023
13	18	17	11	13	18	1013
14	18	17	10	13	18	1010
13	18	16	8	14	18	1009
12	17	16	11	15	17	1002
12	18	15	9	14	16	1000
14	17	16	10	15	16	998
14	17	17	11	13	18	982

Speaker notes

. . . three candidates 90% fit to be wizards. They're mostly pretty good in the other stats, but not so much as to be obviously better suited for some other class, except that that top one looks like a bard to me.

Remember though that this is all very random. It may converge on a good solution faster, or more slowly, and the fitness function may be good or poor at getting just the right mix of alleles.

Now, suppose we want to evolve yet another type of thing. By the Rule of Three, it's about time we . . .

Image: from https://www.authoritydental.org/tooth-extraction-recovery

Speaker notes

. . . extract the common parts into something they all can use, something with a name like, oh, say, . . .

Image: my picture, of my actual license plate!

Speaker notes

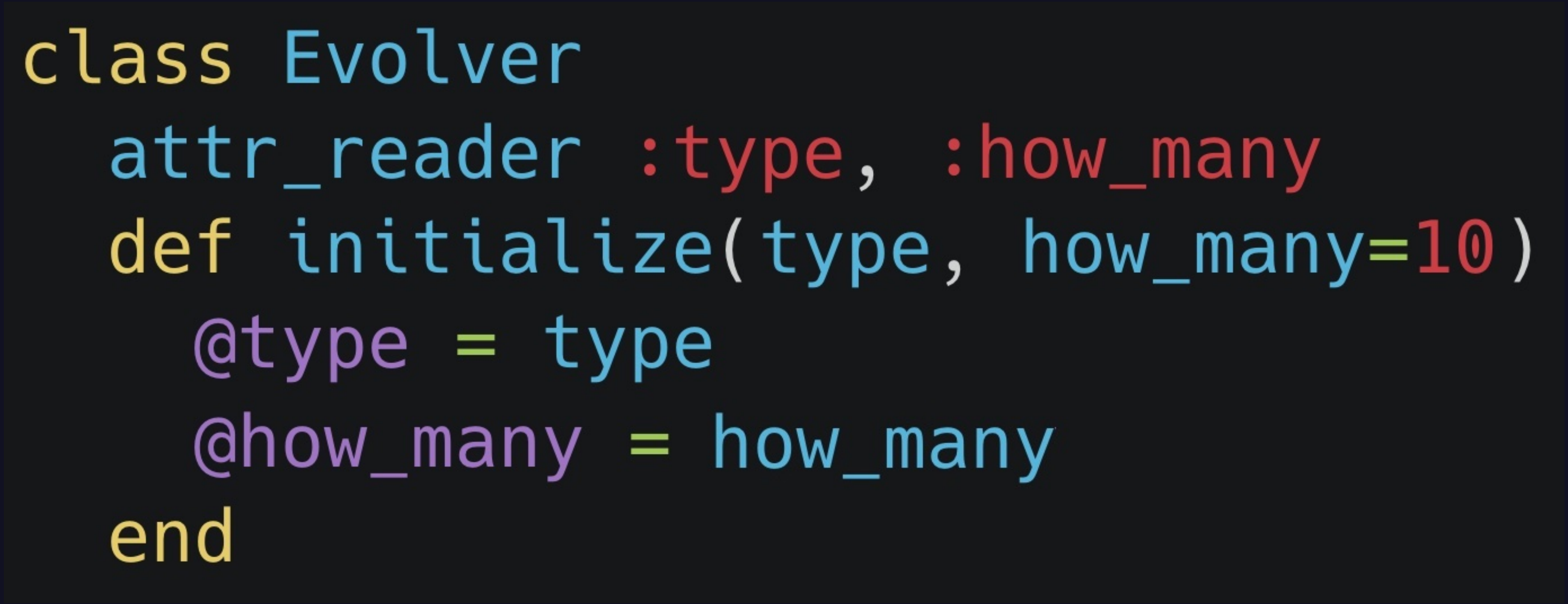
. . . Evolver! That's my actual license plate, by the way. I've had it for several years, since long before I ever tried genetic algorithms, and used that name for this code before I had ever heard of the Evolver PC toolkit that I mentioned near the start.

Anyway, with just a little bit of tweaking, and a slight sprinkling of fancy Ruby magic dust, we can extract a class that defines the common parts, and takes another class that defines the varying parts, and run evolution on it, like so:

```ruby
class Evolver
  attr_reader :type, :how_many
  def initialize(type, how_many=10)
    @type = type
    @how_many = @how_many
  end

  def evolve()
    pop = initial_population().sort_by(&:fitness).reverse
    while not type.done?(pop)
      breeders = type.select_breeders(pop)
      pop = breed(breeders)
      pop.each { |cand| cand.maybe_mutate }
      pop = pop.sort_by(&:fitness).reverse
    end
    return pop
  end

  def initial_population()
    population = []
    for i in 1..how_many
      population.append(type.new)
    end
    return population
  end

  def breed(breeders)
    population = []
    for i in 1..how_many
      population.append(type.combine(breeders))
    end
    return population
  end
end
```

Speaker notes

That's a lot of code to look at at once, so let's go through it piece by piece. Since it is meant to accept a class, and hold onto it for later use, we can use . . .



```
class Evolver
  attr_reader :type, :how_many
  def initialize(type, how_many=10)
    @type = type
    @how_many = how_many
  end
```

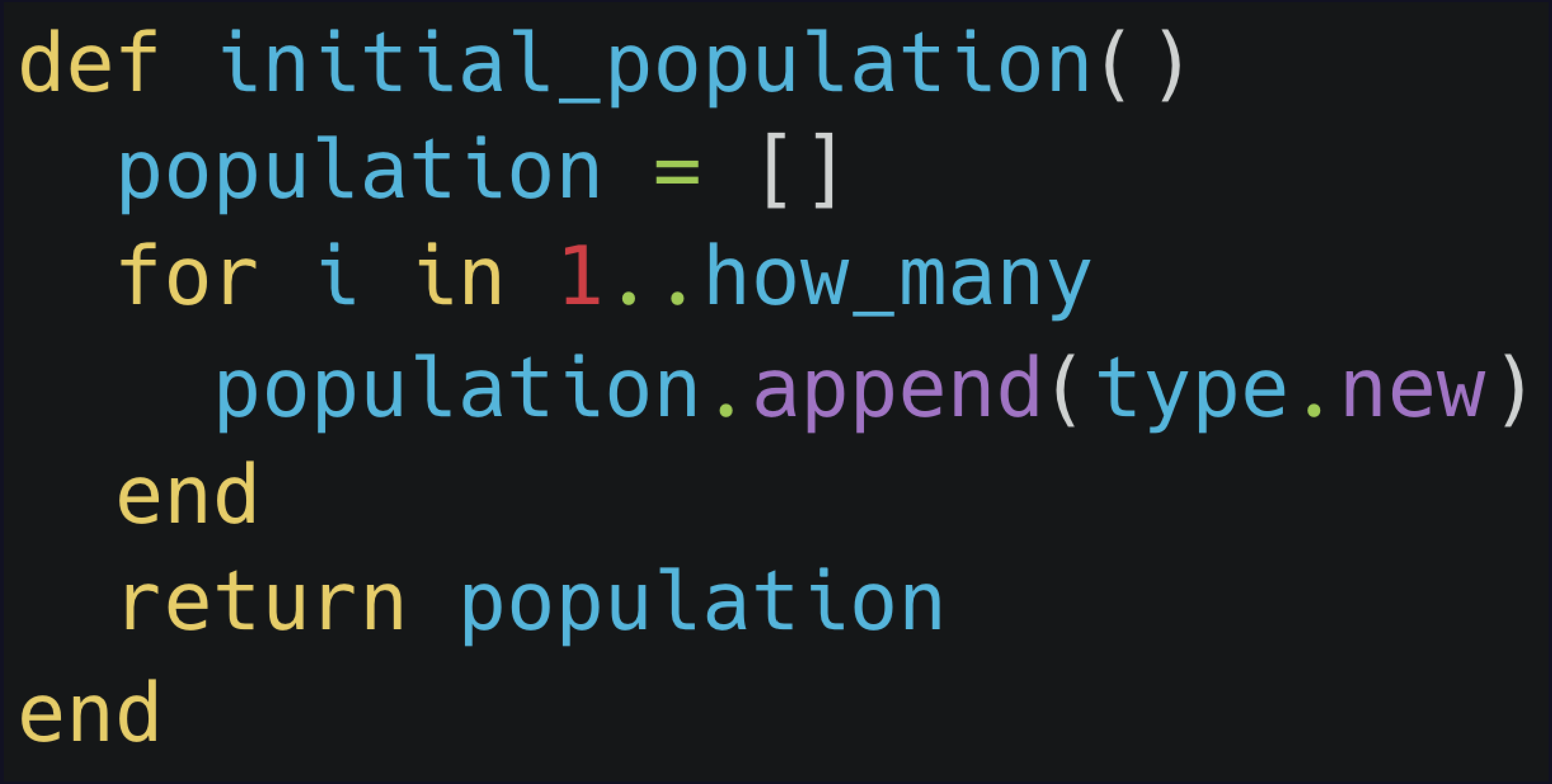
Speaker notes

. . a constructor that just stashes that class. (And also an option for how big a population to use.) How we tell it to "run evolution" can look like this:

```ruby
def evolve()
  pop = initial_population().sort_by(&:fitness).reverse
  while not type.done?(pop)
    breeders = type.select_breeders(pop)
    pop = breed(breeders)
    pop.each { |cand| cand.maybe_mutate }
    pop = pop.sort_by(&:fitness).reverse
  end
  return pop
end
```

Speaker notes

. . . just a slight variation on our original high-level code. Mainly, we're passing the responsibility for determining whether we're done, selecting breeders, and mutating each candidate, over to the passed-in class. We're also sorting the population on descending fitness. Creating the initial population could look like this:



```
def initial_population()
  population = []
  for i in 1..how_many
    population.append(type.new)
  end
  return population
end
```

Speaker notes

. . . nearly identical to our original code, but, again, passing the creation responsibility along to our passed-in class. Breeding a new population could look like this:

```
  def breed(breeders)
    population = []
    for i in 1..how_many
      population.append(type.combine(breeders))
    end
    return population
  end
end
```

Speaker notes

. . . again nearly identical to our original code, but passing the responsibility for how to combine the breeders, over to our passed-in class.

With this Evolver class handling most of the infrastructure, we don't need to repeat most of the boilerplate that we had in both the Truckload and Character classes. We can demonstrate that by evolving something else, and supplying only the parts that vary. So what now? In my spare time, I make . . .

Image: my own pic of five batches of plain mead in progress

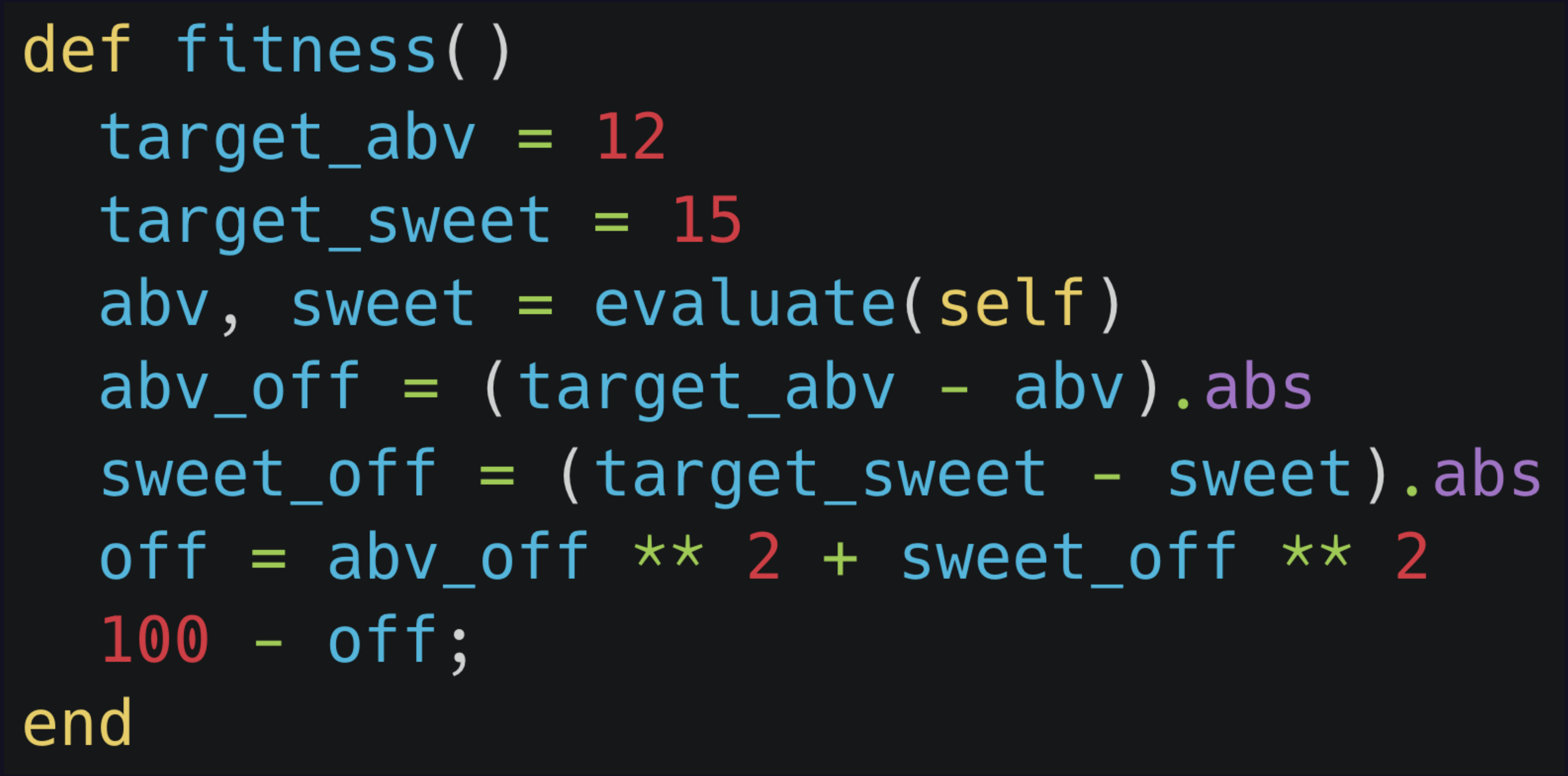
Speaker notes

. . . mead, a wine-like drink made by fermenting honey. Let's see if we can evolve a good recipe. Our recipe will be based on two simple factors:

```ruby
class Recipe
  def initialize()
    # water:honey ratio ranges from 1.5-15
    @ratio = roll(3, 10) / 2.0
    # alcohol tolerance of 6-20
    @tolerance = 4 + roll(2, 8)
  end
end
```

Speaker notes

the ratio of water to honey, and the alcohol tolerance of the yeast we'll use, both by volume. The resulting percent alcohol by volume, or ABV, and the sweetness, are the criteria on which we'll evaluate the recipes, but I'll spare you the details of how we figure them out from the inputs. Beyond simple creation, we'll need a . . .



```
def fitness()
  target_abv = 12
  target_sweet = 15
  abv, sweet = evaluate(self)
  abv_off = (target_abv - abv).abs
  sweet_off = (target_sweet - sweet).abs
  off = abv_off ** 2 + sweet_off ** 2
  100 - off;
end
```

Speaker notes

. . . fitness function. This one looks at how far off we are from our desired target values. We square the error in each aspect separately, so that something that's, say, one point off in each, will score higher than something that's spot-on in one but two points off in the other. In other words, we want them both close, not just trading off one for the other directly.

You can see here we are targeting 12% ABV, out of a range from 0 to 20, and 15 "points" of sweetness, which is on a custom scale . . .

Level	Min	Max
Dry	0	9
Semi-Sweet	10	19
Sweet	20	29
Dessert	30	39
Too Sweet	40	

Speaker notes

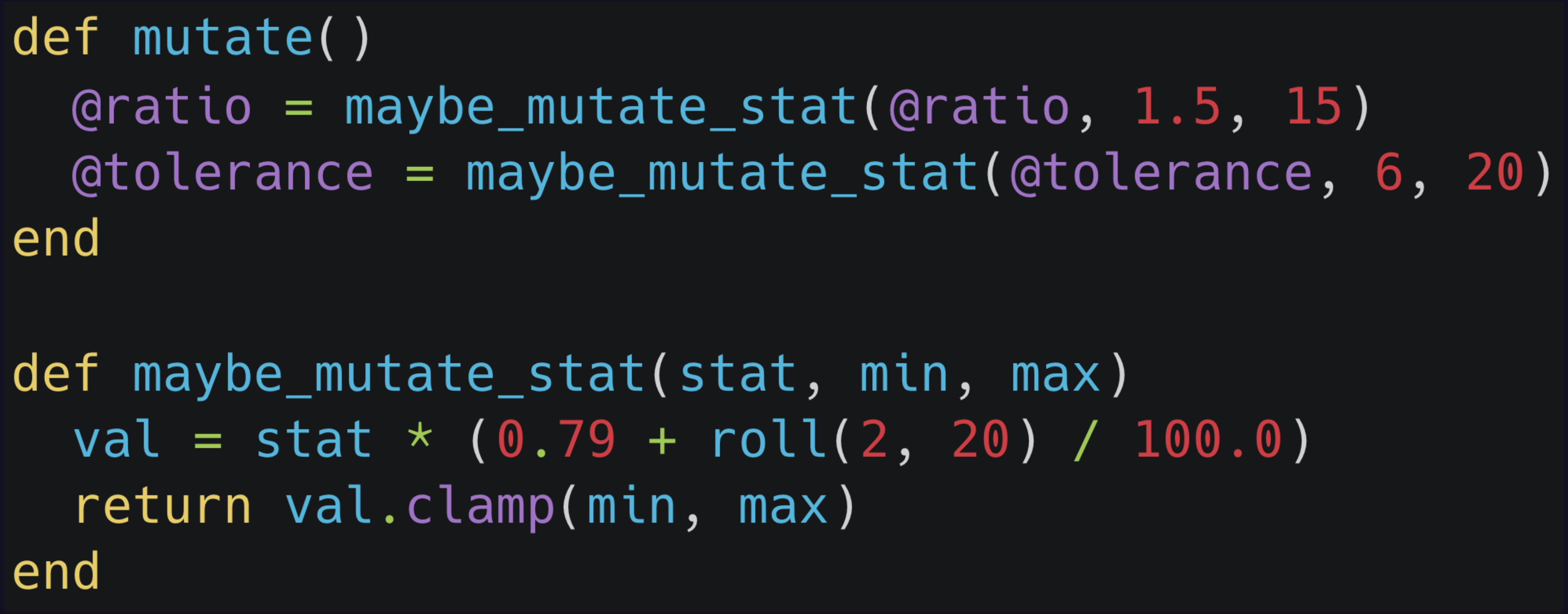
. . . of zero to about 80 for anything reasonable as a finished product. Explaining this chart is beyond the scope of this talk, but if you really want to know, see me later and I'll talk your ears off about mead-making. Anyway, this means that we're going for roughly typical wine strength, and semi-sweet.

We'll stick with the breeder selection function of just taking the top two most fit, but we'll need a combiner function, to combine two parents into one, which could look like this:

```ruby
def self.combine(p1, p2)
  rec = self.new
  rec.ratio     = rand(2) == 1 ? p1.ratio     : p2.ratio
  rec.tolerance = rand(2) == 1 ? p1.tolerance : p2.tolerance
  return rec
end
```

Speaker notes

. . . sticking with the tactic of flipping a coin to see which parent contributes each gene. Once each candidate combination is created, we need a method to mutate it, such as . . .

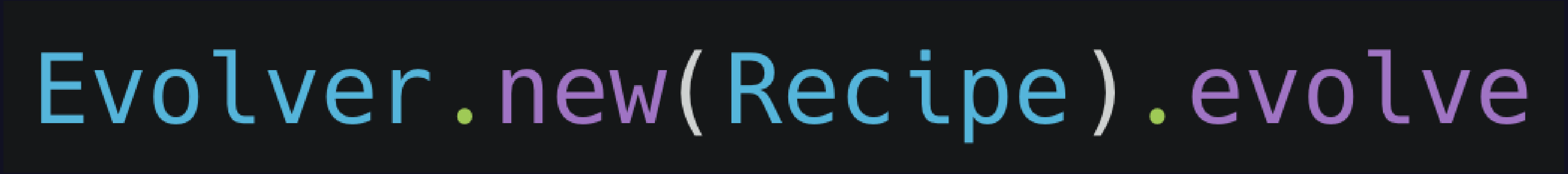


```ruby
def mutate()
  @ratio = maybe_mutate_stat(@ratio, 1.5, 15)
  @tolerance = maybe_mutate_stat(@tolerance, 6, 20)
end

def maybe_mutate_stat(stat, min, max)
  val = stat * (0.79 + roll(2, 20) / 100.0)
  return val.clamp(min, max)
end
```

Speaker notes

. . . this. What we do here is to let each figure go up to 19% either up or down, but clamp it within a reasonable range.

Everything else, such as creating the initial population, selecting breeders, breeding a new population, the concept of how to check if we're done, though with a different threshold, and the overall structure of how it simulates evolution, are all handled by the Evolver class. To actually use them together would look like this:



```
Evolver.new(Recipe).evolve
```

Speaker notes

With our current version of Evolver, this would return the current population of Recipes, when the Recipe class says that it's done.

To demonstrate briefly how it might evolve mead recipes, an initial population might look like this:

Ratio	Tolerance	%ABV	Sweet	Fit
6.50	10.0	7.4	0.0	-145.8
7.00	11.0	7.0	0.0	-150.3
7.50	9.0	6.6	0.0	-154.6
8.00	14.0	6.2	0.0	-158.7
8.50	12.0	5.9	0.0	-162.6
9.00	14.0	5.6	0.0	-166.2
9.00	9.0	5.6	0.0	-166.2
12.50	11.0	4.1	0.0	-186.9
12.50	16.0	4.1	0.0	-186.9
13.00	9.0	4.0	0.0	-189.3

Speaker notes

. . . showing that we can have negative fitness, and the next generation might look like this:

Ratio	Tolerance	%ABV	Sweet	Fit
5.46	10.6	8.6	0.0	-136.3
5.52	11.7	8.5	0.0	-136.9
5.72	9.6	8.3	0.0	-138.7
5.95	9.4	8.0	0.0	-140.8
5.98	10.7	8.0	0.0	-141.1
6.37	10.9	7.6	0.0	-144.6
6.44	11.3	7.5	0.0	-145.2
6.56	12.2	7.4	0.0	-146.4
7.42	9.7	6.6	0.0	-153.9
7.84	9.6	6.3	0.0	-157.4

Speaker notes

. . . with slightly improved fitness and a narrower range, and after just 7 generations, finally this:

Ratio	Tolerance	%ABV	Sweet	Fit
3.46	10.4	10.4	16.4	95.2
3.59	9.9	9.9	17.4	89.5
3.85	8.9	8.9	20.1	64.6
3.42	9.7	9.7	22.5	38.1
4.53	9.2	9.2	6.7	24.0
3.50	9.2	9.2	24.4	4.6
4.33	9.8	9.8	5.3	1.8
3.94	7.7	7.7	27.1	-64.1
4.33	11.0	10.5	0.0	-127.4
3.53	8.3	8.3	30.6	-156.7

Speaker notes

with one candidate weaker and sweeter than we want, but within our 90%-fit criterion. If we get really picky and demand a 99.9% fit, my first run like that took 45 generations, and yielded this:

Ratio	Tolerance	%ABV	Sweet	Fit
3.05	11.8	11.8	15.2	99.9
3.21	10.9	10.9	18.2	88.7
3.34	11.4	11.4	10.8	81.7
3.28	10.5	10.5	19.2	79.8
3.35	11.7	11.7	8.9	62.9
3.31	11.8	11.8	8.8	61.4
3.48	9.7	9.7	21.0	58.9
3.05	13.1	13.1	4.9	-2.9
3.60	11.8	11.8	2.6	-53.9
3.68	12.2	11.9	0.0	-125.0

Speaker notes

. . . a tiny bit weaker and sweeter than asked for.

Just like with the DnD characters, if we want to change what we're after, we just have to tweak the fitness function. This time, I asked for 7% alcohol and sweetness of 5, a low-alcohol dry mead, often called a "session" mead, and still demanding a 99.9% fit, it took 37 generations to get this:

Ratio	Tolerance	%ABV	Sweet	Fit
6.38	6.9	6.9	5.0	100.0
5.73	7.6	7.6	5.4	99.5
5.32	8.2	8.2	4.9	98.6
5.61	7.6	7.6	6.5	97.3
6.25	6.7	6.7	7.8	92.2
5.34	7.7	7.7	8.8	85.5
5.23	8.8	8.8	1.0	80.7
6.25	8.2	7.7	0.0	74.5
6.02	8.3	7.9	0.0	74.1
5.61	6.8	6.8	12.3	46.5

Speaker notes

. . . spot-on in sweetness and just a tiny bit weaker. Then I asked for 16% alcohol and a sweetness of 35, a strong and sweet mead, historically called sack mead or a great mead, a style very popular in Poland. It took 149 generations, which was still very fast, because the functions were so simple, to produce this:

Ratio	Tolerance	%ABV	Sweet	Fit
1.69	16.1	16.1	34.9	100.0
1.70	15.8	15.8	36.9	96.4
2.06	13.6	13.6	35.2	94.3
1.96	14.0	14.0	36.5	94.0
1.94	15.4	15.4	27.3	39.9
1.75	14.6	14.6	43.0	33.8
1.92	13.3	13.3	43.8	16.1
1.68	14.2	14.2	50.6	-146.3
1.93	17.6	17.6	10.6	-497.1
1.70	11.8	11.8	67.8	-992.7

Speaker notes

. . . just a tiny bit stronger and drier than asked for.

There are many other ways you can use genetic algorithms. They can create images, music, and even code. So, think about it, and you might be able to use them for something.

To recap what you've learned here today:

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts

Speaker notes

Genetic Algorithms are optimization heuristics, which is fancy-talk for shortcuts to finding good-enough solutions. They're . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought

Speaker notes

. . . simpler than you probably thought -- all you have to do is create an initial population, and cycle through five simple steps, of assessing their fitness, checking if you're done, picking breeders, breeding them, and mutating the new candidates, over and over until you're done. They . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions

Speaker notes

. . . can use very simple functions, but it . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good rules

Speaker notes

. . . can be tricky to figure out exactly what the functions should do, especially for evaluating fitness and checking if you're done, so as to make the solutions converge quickly enough, and not ignore too many other very good solutions. This approach is also . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good rules
- applicable to a wide variety of problems

Speaker notes

. . . applicable to a huge variety of problems, including ones so complex that . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good rules
- applicable to a wide variety of problems
- can create solutions humans would not

Speaker notes

. . . a semi-random algorithm can come up with excellent solutions that we humans would never have thought of.

Now, if you have any . . .

? ? ? ? ?

T.Rex-2023@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson

Repo and Slides:
github.com/CodosaurusLLC/tight-genes
Codosaur.us/reds/gen-algs-ndc-oslo-23-slides

Speaker notes

. . . questions, I'll take them now, or at the contact info shown up there. As for the other URLs, the Github one is for the code, and slides in HTML, and the other one (as you may have guessed) is for the slides as a PDF, complete with a full script. Any questions?