Life continued…

In the previous project, we created a simulation of single-celled creatures which were capable of growing, moving and reproducing. The next steps are to improve on reproduction, and build in the capacity for cells to interact with one another generally.

Helper functions

In order to interact with their environment, our cells will need to know what's around them. They will presumably have a radius of perception (how far they can see / detect), and we'll need an efficient way to find out what is 'visible' to any given cell.

```
distanceTo(other){
  return p5.Vector.sub(other.p, this.p).mag();
}
roomToGrow(){
  for (let i=0; i<cells.length; i++){
    if (cells[i] != this && this.distanceTo(cells[i]) < 60){return false}
  }
  return true;
}
```

Within the *Cell* class, I've now got a *distanceTo* function which tells me how far I am from a given 'other', which could be another cell, but in general just needs to be something with a *p* attribute (ie, position).

I also have a *roomToGrow* function which looks at all cells in the *cells* array, and gives me *false* if any of them (except itself) is too close. If it doesn't find any that are too close, it returns *true*.

Stop overcrowding

I can use *roomToGrow* immediately to inhibit growth when cells would otherwise be too close:

```
// attempt to reproduce
if (this.age > this.lifespan/2){
  if (random() < fertility && this.roomToGrow()){
    this.reproducing = true;
  }
}
```

Within *update*, I can add *this.roomToGrow()* as a condition for reproduction.

Give each other some space

If cells can tell when they are close to other cells, we can give them a gentle nudge to move away, by directing their velocity in the opposite direction to the displacement vector to the other cell:

```
// move
let neighbour = this.findNeighbour();
if (neighbour != null){
  let displacement = p5.Vector.sub(neighbour.p, this.p);
  this.v = displacement.copy().normalize().mult(-0.5);
}
this.v.rotate(random(-PI/12, PI/12));
```

Within *update* I'm adding in some code that will find a neighbour (just the first one I encounter), and use its displacement to change my velocity. Note that I'm using *normalize()* to find the unit vector in that direction, then *mult* to scale it (opposite direction to the vector to the neighbour, and of size 0.5, which is my default speed for cells).

In order to make this work, however, *findNeighbour* needs to exist. Remember it's fine to make up methods and functions as you go – often it's helpful to work out how a function's output will be used before you write it, so it is fit for purpose.

```
findNeighbour(){
  for (let other of cells){
    if (other != this && this.distanceTo(other) < 60){
      return other;
    }
  }
  return null
}
```

This method, within the *Cell* class, should give me the first cell (that isn't itself) which is close to the cell in question, when such a cell exists.

Bug warning! Running the code at this point will give normal behaviour until reproduction. At this point, all cells stop moving, and none of their children will move either. Before reading ahead, can you work out why this happens?

Edge cases

An edge case in programming or mathematics is a particular scenario which isn't necessarily taken into account in the initial design of a process, proof or program. For instance, we could write code to determine whether a point is inside a given circle by comparing its distance to the centre to the radius of our circle. If it's smaller, the point lies within the circle, and if it's larger, it lies outside. The 'edge case' here is literally the case where the point lies on the 'edge': the boundary between being inside and outside. In trivial cases like this it's usually enough for the designer to make a judgement call: any point on or within the boundary counts as inside, or any point on our outside the boundary counts as outside. This may even be done without thought through the use of $<$ or $\leq$.

Fixing our bug

The cause of our particular bug is an edge case that is somewhat more common (although it wasn't obvious). In general, if you consider how two cells should avoid one another, you probably imagine two cells that are in different positions: surely most of the time that will be perfectly true. In these cases, our code works fine. But what if the two cells are in exactly the same position? Unlikely, you reckon? Not if we've encoded them with inheritance so that their position matches that of their parent. In this case, the displacement vector, which we then merrily scale to form the velocity, is just the zero vector. We can either try to avoid the problem by having children appear slightly offset from parents, or we can fix our avoidance code. Of the two, fixing the avoidance code is probably preferable since we don't need to worry about the potential for some other update to wind up exhibiting the same problems:

```
// move
let neighbour = this.findNeighbour();
if (neighbour != null){
  let displacement = p5.Vector.sub(neighbour.p, this.p);
  if (displacement.mag() == 0){
    displacement = p5.Vector.random2D();
  }
  this.v = displacement.copy().normalize().mult(-0.5);
}
this.v.rotate(random(-PI/12, PI/12));
```

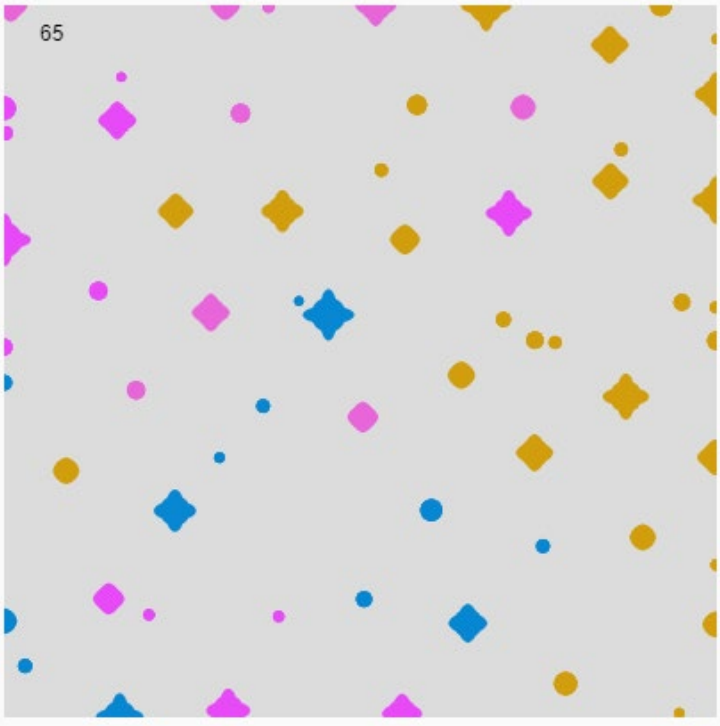
This fix essentially chooses a random direction whenever the displacement is zero, so cells will still move apart (all directions are apart when you're together).

Tracker

```
function draw() {
  background(220);
  fill(0);
  text(cells.length, 20, 20);
```

It may not remain in the final iteration of this program, but for now I'm finding it helpful to know how many cells are on the screen. This puts that number top left at all times.

I left my code running for a while, and this is the result: the equilibrium number of cells appears to be somewhere between 60 and 70. They're jiggling around like they have nowhere to go, which I suppose makes sense given that wherever they go there'll be some other cell to get in the way. New cells can't be generated until old cells die to make room.



Also, a number of cells are bouncing around the edge. To improve on the visual, we could artificially reduce the apparent edge (according to the maximum radius of a cell):

```
this.p.add(this.v);
if (abs(this.p.x - width/2)>width/2 - 15){
  this.v.x *= -1;
  this.p.x += this.v.x;
}
if (abs(this.p.y - height/2)>height/2 - 15){
  this.v.y *= -1;
  this.p.y += this.v.y;
}
```

Refactoring

As you develop and add complexity to a program, you may often find that something that worked fine before is becoming increasingly unfit for purpose. It may be as simple as a variable that you're now using in multiple places, so instead of defining it locally (or just hardcoding a specific value even), you want to make it something you only need to change in one place. Here are a few such improvements I'd make:

```
update(){
  // age and check if still alive
  this.age += 0.1;
  this.size = map(this.age, 0, this.lifespan, 5, 20);
```

So I can use the cell's *size* not only in display but also when checking to see what's close, etc. This line is in *update* so it is recalculated each time the cell ages (if you put it in *constructor* it only gets done once).

```
roomToGrow(){
  let d = this.size * 2;
  for (let i=0; i<cells.length; i++){
    if (cells[i] != this && this.distanceTo(cells[i]) < d){return false}
  }
  return true;
}
```

Now I have *this.size*, I can make use of it in *roomToGrow*, so that the larger cells take up more room.

```
circle(this.p.x, this.p.y, this.size);
noFill();
stroke(this.color);
strokeWeight(0.5);
circle(this.p.x, this.p.y, this.size*2);
```

In the *display* method, I'm now also using *this.size* to draw a circle around the cell: this may or may not make the final cut (they do look more like cells with membranes), but at least for now it helps me see how well they're observing social distancing.

```
let cells = [];
let n = 40;
let maxN = 500;
let fertility = 0.01;
let maxSpeed = 0.5;
```

I'm also giving a value to the maximum speed, to make it easier to modify this further down the line.

```
constructor(parent=null){
  if (parent == null){
    this.p = createVector(random(20, width-20), random(20, height-20));
    this.v = p5.Vector.random2D().mult(maxSpeed);
```

```
// move
let neighbour = this.findNeighbour();
if (neighbour != null){
  let displacement = p5.Vector.sub(neighbour.p, this.p);
  if (displacement.mag() == 0){
    displacement = p5.Vector.random2D();
  }
  this.v = displacement.copy().normalize().mult(-maxSpeed);
```

Notice that this affects the *constructor* method of the *Cells* class and also the code that determines how they cope with being too close to other cells (inside *update*).

Another bug fix

Having changed the behaviour near the boundary, I'm noticing some curious artefacts: cells that are created too close to the edge of the screen never get away from there. This should fix that.

```
class Cell{
  constructor(parent=null){
    if (parent == null){
      this.p = createVector(random(20, width-20), random(20, height-20));
```

Cells, when randomly generated, are done so within a smaller range.

Eat to live, live to eat

It would be wrong for cells to eat their own offspring or parents, but we can build in a way of attempting to consume other life forms.

```
tryToEat(other){
  let [e, E] = [other.energy, this.energy];
  if (e > E){return false}
  if (random() < (E / (e + E))){
    this.energy += other.energy;
    this.energy = min(this.energy, maxEnergy);
    other.alive = false;
  }
  else{
    other.energy += this.energy;
    other.energy = min(other.energy, maxEnergy);
    this.alive = false;
  }
}
```

This method within the *Cell* class lets any cell attempt to consume another life form (all that is required about the other life form is that it also has *energy* and *life* attributes – we could design a new form of life later for our cells to prey on provided we gave it these attributes).
Note that this attempt to eat doesn't take place unless the cell in question has more energy than the other. This means we won't end up with two cells simultaneously eating each other.

We should also define *maxEnergy* at some point: this is used in our code above to ensure we don't end up with super-cells, consuming all in their path and becoming stronger and stronger with a positive feedback loop.

```
let fertility = 0.01;
let maxSpeed = 0.5;
let maxEnergy = 100;
```

```
this.alive = true;
this.reproducing = false;
this.energy = 100;
```

We'll need to actually give our cells an *energy* attribute.

And it makes sense for energy to be depleted. In future iterations perhaps we could have our cell lose more energy when moving fast, but ours move at a constant speed. For now, I'm allowing cells to continue living even on zero energy, but their lifespan will be reduced by 0.1 for every tick when they don't have any energy.

```
update(){
  // age and check if still alive
  this.age += 0.1;
  this.energy -= 0.1;
  this.energy = max(0, this.energy);
  if (this.energy == 0){
    this.lifespan -= 0.1;
  }
```

```
// move... and eat?
let neighbour = this.findNeighbour();
if (neighbour != null){
  let dir = 1;
  if (neighbour.color == this.color){dir = -1}
  else{this.tryToEat(neighbour)}
  let displacement = p5.Vector.sub(neighbour.p, this.p);
  if (displacement.mag() == 0){
    displacement = p5.Vector.random2D();
  }
  this.v = displacement.copy().normalize().mult(dir * maxSpeed);
}
this.v.rotate(random(-PI/12, PI/12));
```

In the *update* method, I'm amending our use of the neighbour we potentially found: if it's the same colour, we just move away from it. If it's a different colour, we try to eat it (and move towards it). For now, colour is how we distinguish between 'families'.

I recommend adding these lines, even if only temporarily, so we can see the relative energy of our different cells.

```
circle(this.p.x, this.p.y, this.size*2);
noStroke();
fill(0);
text(int(this.energy), this.p.x+15, this.p.y);
```

```
let cells = [];
let colors = ["red", "green", "blue"];
let n = 3;
```

If we want to make it a bit clearer what's going on, we can decide in advance the colours of our cells.

And when we first generate them in *setup*, we can set their colours.

```
function setup() {
  createCanvas(400, 400);
  textAlign(CENTER, CENTER);
  for (let i=0; i<n; i++){
    let c = new Cell();
    c.color = colors[i]
    cells.push(c);
  }
}
```

What's next?

Instead of just showing the total number of cells at the top, can you build in a tally for the number of red, green and blue cells present? Maybe even a miniature bar graph in one corner, so we can see how the numbers fluctuate over time.

Consider energy conservation in our system: with the current set-up, offspring cost no energy (in fact, they are a net gain, since they always come with 100 energy by default). Maybe cells could gain energy through consumption of others, but also by, say, ending up near a basic fixed food source (eg corners?) Maybe it should cost some energy to have offspring: you can only reproduce by spending 30, and only if you have 30 to spend.