1. MOTIVATION: WHY OOP?
   a. **Design:** Makes the choices of designing a program (particularly the choices of where to divide) more intuitive.
   b. **Re-use:** Makes code re-use sometimes even easier.
   c. **Encapsulation:** Ties together data-structures with the processes that operates on those data structures into one package (this is key - more on this in the next class).
   d. **Data abstraction:** Details of the implementation are hidden from the user - including details of representation of the data structure.
   e. **Real-world modeling:** Developed in the 50s, 60s, at MIT, within the context of people working in Artificial Intelligence.
   f. **THAT SAID,** object-oriented programming is simply a way of thinking about problems.
   g. **As such,** it’s only one tool in your problem-solving toolbox.
   h. **And it doesn’t always need to be used!** If the logic of what you are trying to program is straightforward, and an object-oriented program does not seem intuitive or necessary, then you shouldn’t force yourself to use it.
   i. **History:** 50s, 60s, MIT, within the context of people working in Artificial Intelligence. Blew up in the 90s.
   j. **Nowadays** (for the last 10 years or so), OOP gets often taught as the very basics of programming itself. This can be limiting, constraining, and absolute nonsense.
   k. **I think it’s important** to understand the basics of programming outside of OOP - the bare bones. And have OOP be one more tool that you make a deliberate choice to use, not the default.

2. FUNDAMENTALS OF AN OBJECT
   a. An object has two features:
      i. **It has attributes:** the data structures that define that class, and make each instance difference.
      ii. **It also has methods:** the types of operations that are allowed for that class.

3. DEFINING A CLASS:
   a. Revisiting the `point` class example:

```
class point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b
    def move(self, a, b):
        self.x += a
        self.y += b
    def __str__(self):
        return "(%d,%d)" % (self.x, self.y)
```

4. ADDING MORE METHODS
   a. We can add a method that calculates the *distance* to the origin.
      i. Notice the difference between creating an attribute called distance, that always gets updated, and creating a method that calculates it.
   b. We can add a method that re-initializes the point’s location to 0.

5. ADDING SOME RANDOMNESS
   a. We can add a method that *reinitializes* the point randomly to a new orientation. Or we can actually chance the existing re-initializing method by a concept called method overloading.
   b. We can add a method that *displaces* the point to a random point in the vicinity of the current location.

6. CALLING METHODS WITHIN METHODS
   a. Modifying the move method so that it creates a boundary in the world. If out of the boundary, then it reinitializes the point. Call to self.dist() and self.reinit().
b. Another version could stick the point to the specific boundary where it left.

7. CONTROL OVER THE OBJECT
   a. Having the methods that operate on the object means that you have at all times control over the state of the object.
   b. For example, you can make sure the point is always a float.
   c. You can make sure the point is always within some boundary.
   d. This creates a really useful safety in your code.

8. INTERACTION BETWEEN DIFFERENT INSTANCES OF THE SAME OBJECT
   a. We can add a method that calculates the distance between itself, and some other point.

9. USING THE OBJECT IN THE REST OF YOUR CODE
   a. Create a function that creates a point, and displaces it $D$ times. Then check how far did the point end up.
   b. Create a function that repeats that $R$ times. Now check the average distance the object ended up at.
   c. We can easily write some of the results to a file and plot it in Matlab.

 Probably next week, or the week after that:

10. USING THE CLASS AS THE BUILDING BLOCK FOR OTHER CLASSES: RECTANGLE

11. DATA ABSTRACTION: DIFFERENT IMPLEMENTATIONS OF THE SAME OBJECT