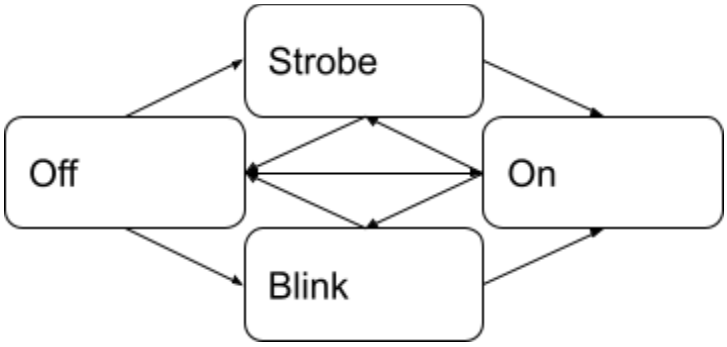
State Machines

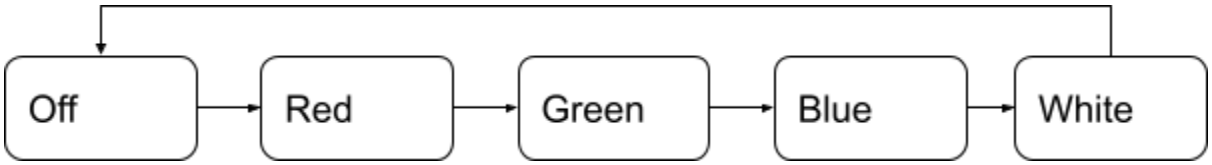
State machines are a common way of representing a program that has multiple **states**. A state is a specific way of being, which we use to describe steps in a longer set of instruction. Most of the code that we have done has been fairly simple, so we have avoided needing to use state machine design tips.

Most arduino code has some kind of state. A blinking light needs to know whether it is in the **on** or **off** state, so we use a variable to keep track of that. State machines work really well for situations where there are a lot of independent states. For example, a project that uses a remote to change how a string of lights blink. The lights can be **off**, **on**, **flashing**, or **pulsing**, each of which is a state. To start, we'll draw a diagram.



This diagram is fairly simple, but it shows that all of our states connect to each other. If we wanted to, we could label each of the arrows with what causes the switch, but it's not necessary.

Another diagram shows a light that switches through 5 color states. This diagram makes it clear which state leads to which.



The easiest way to keep track of state is to use an **Enum**, a special custom variable. To make an enum, you define it(probably at the top of your file) something like this:

```
enum light_state {
  led_on,
  led_off,
  led_blink,
  led_strobe
};
```

`enum` tells the compiler that the variable name `light_state` is not a specific variable, but instead a new **TYPE** of variable. The names inside of the {...} are the states that can be in `light_state`. So, to use the `enum`, we define a new variable like this: `light_state currentState`. Where currentState can be any of the things inside the {...}. Then we can say `currentState = led_on`, or `if(current_state == led_strobe)`, and it will work.

State Machines

This lets us keep track of state much easier than remembering that int state = 1 means on, and int state = 0 is off.