Reactor

Our reactor program takes IR readings and makes very simple decisions accordingly.

```
def ir_callback(self, ir):
    vals = [x.value for x in ir.readings]
    vals = enumerate(vals)
```

First, we use list comprehension to create a list of primitive values from the `IrIntensityVector`. Then we apply the Python `enumerate` function to the list, creating a list of tuples that look like

```
("Direction", "Reading Value")
```

We know that a "Direction" value of 0 indicates a reading from the leftmost sensor, and 6, a reading from the rightmost scanner.

```
threats = list(filter(lambda x : x[1] > 150, vals))
```

We apply a `filter` function to the `vals` list which we have created, considering readings above `150` to be "threats".

```
if len(threats) > 0:
        threat_direction = list(functools.reduce(lambda a,b : a if a[1] >
b[1] else b, threats))
```

We then apply a `reduce` function to this list of threats to find which sensor is reading the highest value from amongst the sensors which *do* detect a threat.

```
turn = Twist()
    match threat_direction[0]:
        case 0:
            turn.angular.z = -.4
        case 1:
            turn.angular.z = -.7
        case 2:
            turn.angular.z = -.8
        case 3:
            turn.angular.z = -1.0
        case 4:
            turn.angular.z = -.8
        case 5:
```

```
            turn.angular.z = -.7
        case 6:
            turn.angular.z = -.4
    self.ir_publisher.publish(turn)
```

We use switch statements to determine the direction we need to turn according to our readings. We only turn clockwise in the presence of a threat so we don't have to deal with getting stuck in "saddle points."

```
 else:
        forward = Twist()
        forward.linear.x = .15
        turn_direction = list(functools.reduce(lambda a,b : a if a[1] >
b[1] else b, threats))
        forward.angular.z = .05 if turn_direction[0] > 3 else -.05
        self.ir_publisher.publish(forward)
```

If we don't see a threat, then we just publish a forward velocity. We want to turn away from potential threats, so we use a `reduce` function to determine which sensor is receiving the largest reading.

We turn counter-clockwise slowly if the greatest reading is coming from the right, and clockwise otherwise.

Wanderer

Our wanderer program is much more complicated, thanks to the handy programming paradigm of **timers**. Let's take a look at our constructor, specifically our member variables.

```
 def __init__(self):
    super().__init__('wanderer')

    self.hazard_subscription = self.create_subscription(
        HazardDetectionVector,
        'qbert/hazard_detection',
        self.hazard_callback,
        qos.qos_profile_sensor_data)

    self.hazard_subscription  # prevent unused variable warning
    self.hazard_publisher = self.create_publisher(Twist,
"qbert/cmd_vel",10)
    # timer for each movement type
    self.back_timer = None
    self.turn_timer = None
    self.forward_timer = None
    # timer to stop timers
    self.timer_stopper = None
    # boolean ensures only one process runs at a time
    self.executing = False
    # need two delay timers for our algorithm
```

```
        self.delayer_a = None
        self.delayer_b = None
```

We have six timer variables; three of our timers are responsible for different movements associated with `qbert`.

We also have a timer responsible for stopping timers.

We maintain `executing` to make sure our wanderer program doesn't allow itself to overlap turning operations.

Finally, our `delayer` timers are utilized to buffer each sequential movement in our wanderer program.

```
def hazard_callback(self, haz):
    if len(haz.detections) > 0:
        types = [val.type for val in haz.detections] #list comprehension
extracts types from list of objects
        if (1 in types) and not self.executing: #type 1 is collision
```

The hazard vector we're subscribed to maintains multiple hazard varieties. We only care about `BUMP`s. We use list comprehension to extract the types of hazards that `qbert` is publishing, then, so long as we detect a `BUMP`, and we aren't already *wandering*, we begin our wanderer program.

```
 if(self.forward_timer):
    self.destroy_timer(self.forward_timer) #stop going forward!!!
    self.executing = True
    print("bumped")
    self.hazard_publisher.publish(Twist())
    self.back_timer = self.create_timer(.1, self.back_callback)
    self.timer_stopper = self.create_timer(2, self.destroy_back)
```

These are the first lines that get executed after we detect a `BUMP`. We destroy the timer maintaining `qbert`'s forward movement, then we tell our `node` that we are in the process of turning.

We publish an empty `Twist`, telling `qbert` to stop, then we initialize the `back_timer`, which will tell `qbert` to start backing up slowly. We also initialize a timer, `timer_stopper`, to stop `back_timer`, in `2` seconds.

```
 self.delayer_a = self.create_timer(2, self.delayed_turn)
```

Here is where our program trace becomes harder to follow. This next line tells `qbert` to perform a delayed turn in `2` seconds. Let's look at `delayed_turn`.

```
def delayed_turn(self):
    turn_time = random.uniform(4,5)
    self.turn_timer = self.create_timer(.1, self.turn_callback)
    self.timer_stopper = self.create_timer(turn_time, self.destroy_turn)
    self.destroy_timer(self.delayer_a)
```

`turn_time` is a uniform continuous random variable, which will determine how long `qbert` will turn for. We initialize the `turn_timer`, which sustains `qbert`'s turning motion interminably. `timer_stopper` is responsible for destroying `turn_timer` in `turn_time` seconds. Finally, because `delayer_a` is intended to be a *one-shot* timer, we destroy `delayer_a` here.

Almost done, let's look at the last component of our turning program.

```
self.delayer_b = self.create_timer(7, self.delayed_forward)
```

`delayer_b` initializes forward movement 7 seconds into our turning program. `delayed_forward` initializes `forward_timer`, and notice that we only destroy `forward_timer` upon initializing our turning program.

```
elif not self.executing:
        forward = Twist()
        forward.linear.x = .1
        self.hazard_publisher.publish(forward)
```

If we don't detect any bumps, we just publish a forward velocity command.

Thus concluding our wanderer program.