class TBTurnSmoothly:
    def __init__(self, theta, k = 1.25):
        # remember the turning angle
        self.turn = theta
        # the gain for the controller
        self.k = k

def reset(self):
    (sonars, pose) = collectSensors()
    # initialize the initial and the current angle

def step(self, sensors):
    (sonars, pose) = sensors
    # update the error
    self.error = self.target - pose[2]
    return self.currentOutput()

def currentOutput(self):
    if not self.done():
        # specify the forward and rotational velocity
        return (0.0, self.k*self.error)
    else:
        # stop
        return (0, 0)

def done(self):
    # test we have rotated the desired angle
    return abs(self.error) < .01

The only real change is in the currentOutput method, which before used to use a constant turning rate.

**Question 2 (10 points):** The basic equations are:

\[
\Theta[n] = \Theta[n - 1] + \delta_t \Omega[n - 1]
\]

\[
\Omega[n] = K(\Theta_{des}[n - 1] - \Theta[n - 1])
\]

The operator version is:

\[(1 - R)\Theta = \delta_t R\Omega\]

\[\Omega = KR(\Theta_{des} - \Theta)\]
Combining we get:

$$\theta[n] - \theta[n-1] + K\delta_T \theta[n-2] = K\delta_T \theta_{des}[n-2]$$

Equivalently:

$$1 - R\theta + K\delta_T R^2 \theta = K\delta_T R^2 \theta_{des}$$

We accepted a number of variations.

**Question 3 (10 points):** The system function is:

$$\frac{\theta}{\theta_{des}} = \frac{K\delta_T R^2 \theta_{des}}{1 - R\theta + K\delta_T R^2 \theta}$$

Substitute $R = 1/z$ in the denominator polynomial and we get:

$$z^2 - z + K\delta_T$$

The roots of this polynomial are the poles:

$$1 \pm \sqrt{1 - 4K\delta_T}$$

Note that for $K = 0$, one of the poles is 1.0. For $K < 0$, one of the poles is always greater than 1.0. This makes sense since for $K < 0$, the feedback increases the error.

For $\delta_T = 0.2$, for $K = 5$, we have a pole of magnitude 1.0 and they get bigger with bigger gain. The range of stable $K$ is $0 < K < 5$

For $\delta_T = 0.05$, for $K = 20$, we have a pole of magnitude 1.0 and they get bigger with bigger gain. The range of stable $K$ is $0 < K < 20$

In general, when $4K\delta_T > 1$ we have complex poles. For monotonic convergence we need positive real poles, so we want:

$$0 < K \leq 1/(4\delta_T)$$

Note that when $K = 1/(4\delta_T)$, we have the lowest magnitude pole, the pole approaches 1 as $K$ decreases towards 0, so:

For $\delta_T = 0.2$, the best value of $K$ is 1.25, which leads to a pole of $1/2$. For $\delta_T = 0.05$, the best value of $K$ is 5.0, which also leads to a pole of $1/2$.

**Question 4 (5 points):** The analysis above shows that when $\delta_T$ increases the range of stable gains decreases. As the $\delta_T$ increases we need smaller and smaller gains to keep monotonic convergence. The result is very sluggish performance. In general, less frequent observations of the actual state of the robot (bigger $\delta_T$) hurts performance and can lead to instability.
**Question 5 (3 points):** The original implementation of TBTurn used a constant velocity of motion and a constant threshold to detect termination. We had to pick the threshold so that we would not miss the termination, that is, the threshold had to be bigger than $\Omega_\delta T$. So, the bigger the velocity, the bigger the final orientation error we had to put up with. And, this error accumulated as we made more moves.

**Question 6 (5 points):** This is like the square we did in lab.
```python
def setup():
    robot.behavior = SequentialTSM([TBDriveSmoothly(1),
                                     TBTurnSmoothly(2*math.pi/3),
                                     TBDriveSmoothly(1),
                                     TBTurnSmoothly(2*math.pi/3),
                                     TBDriveSmoothly(1)])
    robot.behavior.reset()

def step():
    (fvel, rvel) = robot.behavior.step(collectSensors())
    motorOutput(fvel, rvel)
```

Note that we need turn the **exterior** angle of the triangle. But, we didn’t take points off for that.

**Question 7 (7 points):** Here are the equations for this system:

\[
\begin{align*}
\Theta[n] &= \Theta[n-1] + \delta_T \Omega[n-1] \\
\Omega[n] &= \Omega[n-1] + \delta_T \Xi[n-1] \\
\Xi[n] &= K_1 E[n] + K_2 E[n-1] \\
E[n] &= \Theta_{des}[n] - \Theta[n]
\end{align*}
\]

where $\Xi$ is the acceleration. So, basically, the velocity is the integrated acceleration and the position is the integrated velocity. And, the acceleration is given by the gains times the position errors.
```python
def robot(k1, k2):
    dt = 0.2
    control = systemFunctionFromDifferenceEquation([1], [k1, k2])
    vel = systemFunctionFromDifferenceEquation([1, -1], [0, dt])
    pos = systemFunctionFromDifferenceEquation([1, -1], [0, dt])
    rob = control.cascade(vel).cascade(pos).feedback()
    return max(map(abs, rob.poles()))
```

Depending on the resolution of the search, one gets very different answers.

\begin{align*}
\text{minOverGrid}(\text{robot}, -10, 10, -10, 10, 1, 1) &\approx (0.9999999999999996, (1, -1)) \\
\text{minOverGrid}(\text{robot}, -10, 10, -10, 10, 0.5, 0.5) &\approx (0.8461798864100905, (2.5, -2.0)) \\
\text{minOverGrid}(\text{robot}, -10, 10, -10, 10, 0.25, 0.25) &\approx (0.7499999999999933, (1.75, -1.5)) \\
\text{minOverGrid}(\text{robot}, -10, 10, -10, 10, 0.1, 0.1) &\approx (0.688634457665866, (1.499999999999816, -1.30000000000018))
\end{align*}
Question 8 (10 points): There are many ways of writing this. Here are a few.

```python
def argmax(elements, f):
    bestScore = None
    bestElement = None
    for e in elements:
        score = f(e)
        if bestScore == None or score > bestScore:
            bestScore = score
            bestElement = e
    return bestElement
```

```python
def argmax(elements, f):
    bestElement = elements[0]
    for e in elements:
        if f(e) > f(bestElement):
            bestElement = e
    return bestElement
```

```python
def argmax(elements, f):
    vals = [f(e) for e in elements]
    return elements[vals.index(max(vals))]
```

```python
def argmax(elements, f):
    return max(elements, key=f)
```

Question 9 (5 points): Here are a couple of solutions that work:

WOPQ([Fish.length, Fish.width], [0.9, 0.1])
WOPQ([lambda x: x.length(), lambda x: x.width()], [0.9, 0.1])

Question 10 (10 points): We need to define the priority for an element, then we select the maximum (using argmax), remove and return it.

```python
def extract(self):
    def priority(e):
        return sum([w*f(e) for (w,f) in zip(self.weights, self.features)])
    best = argmax(self.elements, priority)
    self.elements.remove(best)
    return best
```

Question 11 (5 points): Sydney is the longest fish and the priority weights heavily favor length.

wopq.extract() = Sydney

Question 12 (10 points): We need to define the priority function and then just use FPQ.

```python
class WOPQ(FPQ):
    def __init__(self, features, weights):
        def priority(e):
            return sum([w*f(e) for (w,f) in zip(weights, features)])
        FPQ.__init__(self, priority)
```

That’s it... the other methods are provided by the FPQ class.
Question 13 (10 points):  This is relatively easy using the key in the sorted function.

```python
def featureRange(feature, elements):
    vals = [feature(e) for e in elements]
    return max(vals) - min(vals)

def selectFeatures(features, n, elements):
    return sorted(features, key = featureRange)[-n : ]
```