```python
# Import Statements
import numpy as np
from scipy import stats
from tqdm import tqdm
import matplotlib.pyplot as plt
import click

# Creating R.V. X, Y
# For Y, YF and YC are for the outcomes of fair and cheating rolls

class hidden_cas():
    def __init__(self,a,YF,YC,T,b,n):
        # a is probability of changing casino state
        self.a = float(a)
        self.YF = stats.rv_discrete(name = "YF",values = ((1,2,3,4,5,6), YF))
        self.YC = stats.rv_discrete(name = "YC",values = ((1,2,3,4,5,6), YC))
        # Case 0 = Fair, 1 = Cheat
        self.cas_start = 0
        self.burnin = b
        self.n_iter = n
        self.T = T
        self.M = np.asmatrix([[1-self.a, self.a],[self.a, 1-self.a]])
        self.emission = np.asmatrix([YF,YC])
        self.xc,self.yc = self.build_instances()
        self.mcmc_state = self.mcmc()
        self.alphaF, self.alphaC = self.forward()
        self.betaF,self.betaC = self.backward()
        self.Z = self.find_Z()

    def build_instances(self):
        # For part a
        # Take initial state, do t iterations
        # xc and yc are states and rolls at each step i
        xc = [self.cas_start]
        y0 = self.YF.rvs(1)-1
        yc = [y0]
        cstate = 0
        for i in tqdm(range(self.T)):
            p = np.random.uniform(0,1)
            if p < self.a:
                cstate = 1-cstate
            xc.append(cstate)
            if cstate == 0:
                yc.append(self.YF.rvs(1)-1)
            elif cstate == 1:
                yc.append(self.YC.rvs(1)-1)
        return xc,yc

    def find_Z(self):
        Z = np.dot(self.alphaF,self.betaF)+np.dot(self.alphaC,self.betaC)
        return Z

    def sample(self,t):
        return self.xc[t-1],self.yc[t-1]

    def mcmc_flip(self,cstate,ostate):
        cstate = cstate
        pstate = 1-cstate
        ostate = ostate
        if np.random.uniform(0,1) < self.M[ostate,pstate]:
            cstate = pstate
        return cstate

    def prob_cheat(self,chain):
        cc = chain.count(1)
        return cc/(len(chain))

    def mcmc_prob(self,chain):
        p = 1
        for i in range(len(chain)):
            p   *= self.M[chain[i-1], chain[i]]*self.emission[chain[i],self.yc[i]-1]
        return p


    def mcmc(self):
        cstate = 0
        mcmcstates = [0]
        for i in tqdm(range(self.T)):
            p = np.random.uniform(0,1)
            if p < self.a:
                cstate = 1-cstate
            mcmcstates.append(cstate)

        for i in tqdm(range(self.burnin)):
            mcstp = mcmcstates.copy()
            r = np.random.randint(1,200)
            if r == 1:
                mcstp[0] = 0
            else:
                mcstp[r-1] = self.mcmc_flip(mcmcstates[r-1],mcmcstates[r-2])

            acceptance = min(1,(self.mcmc_prob(mcstp)/self.mcmc_prob(mcmcstates)))

            if np.random.uniform(0,1) < acceptance:
                mcmcstates = mcstp

        for i in tqdm(range(self.n_iter)):
            mcstp = mcmcstates.copy()
            r = np.random.randint(1,200)
            if r == 1:
                mcstp[0] = 0
            else:
                mcstp[r-1] = self.mcmc_flip(mcmcstates[r-1],mcmcstates[r-2])

            acceptance = min(1,(self.mcmc_prob(mcstp)/self.mcmc_prob(mcmcstates)))

            if np.random.uniform(0,1) > acceptance:
                mcmcstates = mcstp
        return mcmcstates

    def forward(self):
        alphaF = [self.YF.pmf(self.yc[0])]
        alphaC = [0]
        for i in tqdm(range(self.T)):
            aF = self.M[0,0]*alphaF[i-1]*self.emission[0,self.yc[i]-1] + self.M[0,1]*alphaC[i-1]*sel
            f.emission[1,self.yc[i]-1]
            aC = self.M[1,0]*alphaF[i-1]*self.emission[0,self.yc[i]-1] + self.M[1,1]*alphaC[i-1]*sel
            f.emission[1,self.yc[i]-1]
            alphaF.append(aF)
            alphaC.append(aC)

        return alphaF,alphaC

    def backward(self):
        betaFd = [1]
        betaCd = [1]
        for i in tqdm(range(self.T)):
            bF = self.M[0,0]*betaFd[i-1]*self.emission[0,self.yc[i]-1] + self.M[0,1]*betaCd[i-1]*sel
            f.emission[1,self.yc[i]-1]
            bC = self.M[1,0]*betaFd[i-1]*self.emission[0,self.yc[i]-1] + self.M[1,1]*betaCd[i-1]*sel
            f.emission[1,self.yc[i]-1]
            betaFd.append(bF)
            betaCd.append(bC)
        betaF = betaFd[::-1]
        betaC = betaCd[::-1]
        return betaF, betaC

    def t_is_cheat(self,t):
        return self.betaC[t-1]*self.alphaC[t-1]*(1/self.Z)

    def plots(self):
        z = []
        yfb = []
        ya = []
        ymcmc = []
        ymcmcs = []
        for i in tqdm(range(self.T)):
            z.append(i)
            yfb.append(self.t_is_cheat(i-1))
            ya.append(self.xc[i-1])
            ymcmc.append(self.prob_cheat(self.mcmc_state[0:i+1]))
            ymcmcs.append(self.mcmc_state[i-1])

        fig, ax1 = plt.subplots()

        ax2 = ax1.twinx()
        ax1.plot(z,yfb,'b*',label = "Forward/Backward")
        ax2.plot(z,ya,'r--',label = "Actual States")
        ax2.plot(z,ymcmc,'g o',label = "MCMC")
        ax1.set_xlabel('Time')
        ax1.set_ylabel('Probability of Cheating')
        ax2.set_ylabel('State in True Chain')
        ax1.legend(loc='upper right', bbox_to_anchor=(0.5, 1.15), fancybox=True, shadow=True)
        ax2.legend(loc='upper left', bbox_to_anchor=(0.5, 1.15), fancybox=True, shadow=True)
        ax1.set_title("Comparing F/B, MH predictions to Actual Casino")
        plt.savefig('hidden_casino.png')

        fig2, l2 = plt.subplots()
        l2.plot(z,ya, 'r--',label = "State in True Chain")
        l2.plot(z,ymcmcs,'b--',label = "MCMC States")
        l2.legend(loc='upper right', bbox_to_anchor=(0.5, 1.20), fancybox=True, shadow=True)
        l2.set_title("Comparing MCMC State to True State")
        plt.savefig('mcmc_compare.png')


@click.command()
@click.option(
    '--a',
    type = float,
    default=.05,
    show_default=True,
    help='Probability of switching between Cheat and Fair'
)
@click.option(
    '--yf',
    type = np.array,
    default=(1/6,1/6,1/6,1/6,1/6,1/6),
    show_default=True,
    help='Probability of Fair Die'
)
@click.option(
    '--yc',
    type = np.array,
    default=(19/100,19/100,19/100,19/100,19/100,1/20),
    show_default=True,
    help='Probability of Cheat Die'
)
@click.option(
    '--T',
    default=200,
    show_default=True,
    help='T number of observed rolls'
)
@click.option(
    '--T',
    default=200,
    show_default=True,
    help='T number of observed rolls'
)
@click.option(
    '--b',
    default=10000,
    show_default=True,
    help='Burn in iterations'
)
@click.option(
    '--n',
    default=200,
    show_default=True,
    help='MCMC iterations'
)


def main(a,yf,yc,t,b,n):
    casino = hidden_cas(a,yf,yc,t,b,n)
    casino.plots()


if __name__ == "__main__":
    plt.ion()
    main()
```