```
import abc
import enum
import numpy as np
from scipy.stats import binom
from scipy.stats import norm


class PricingEngine(object, metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def calculate(self):
        """A method to implement a pricing model.

           The pricing method may be either an analytic model (i.e.
           Black-Scholes), a PDF solver such as the finite difference metho
           or a Monte Carlo pricing algorithm.
        """
        pass

class BinomialPricingEngine(PricingEngine):
    def __init__(self, steps, pricer):
        self.__steps = steps
        self.__pricer = pricer

    @property
    def steps(self):
        return self.__steps

    @steps.setter
    def steps(self, new_steps):
        self.__steps = new_steps

    def calculate(self, option, data):
        return self.__pricer(self, option, data)


def EuropeanBinomialPricer(pricing_engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, volatility, dividend) = data.get_data()
    steps = pricing_engine.steps
    nodes = steps + 1
    dt = expiry / steps
    u = np.exp(((rate - dividend) * dt) + volatility * np.sqrt(dt))
```

```
    d = np.exp(((rate - dividend) * dt) - volatility * np.sqrt(dt))
    pu = (np.exp((rate - dividend) * dt) - d) / (u - d)
    pd = 1 - pu
    disc = np.exp(-rate * expiry)
    spotT = 0.0
    payoffT = 0.0

    for i in range(nodes):
        spotT = spot * (u ** (steps - i)) * (d ** (i))
        payoffT += option.payoff(spotT)  * binom.pmf(steps - i, steps, pu)
    price = disc * payoffT

    return price


def AmericanBinomialPricer(pricingengine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, volatility, dividend) = data.get_data()
    steps = pricingengine.steps
    nodes = steps + 1
    dt = expiry / steps
    u = np.exp(((rate - dividend) * dt) + volatility * np.sqrt(dt))
    d = np.exp(((rate - dividend) * dt) - volatility * np.sqrt(dt))
    pu = (np.exp((rate - dividend) * dt) - d) / (u - d)
    pd = 1 - pu
    disc = np.exp(-rate * dt)
    dpu = disc * pu
    dpd = disc * pd

    Ct = np.zeros(nodes)
    St = np.zeros(nodes)

    for i in range(nodes):
        St[i] = spot * (u ** (steps - i)) * (d ** i)
        Ct[i] = option.payoff(St[i])

    for i in range((steps - 1), -1, -1):
        for j in range(i+1):
            Ct[j] = dpu * Ct[j] + dpd * Ct[j+1]
            St[j] = St[j] / u
            Ct[j] = np.maximum(Ct[j], option.payoff(St[j]))

    return Ct[0]
```

```
class MonteCarloEngine(PricingEngine):
    def __init__(self, replications, time_steps, pricer):
        self.__replications = replications
        self.__time_steps = time_steps
        self.__pricer = pricer

    @property
    def replications(self):
        return self.__replications

    @replications.setter
    def replications(self, new_replications):
        self.__replications = new_replications

    @property
    def time_steps(self):
        return self.__time_steps

    @time_steps.setter
    def time_steps(self, new_time_steps):
        self.__time_steps = new_time_steps

    def calculate(self, option, data):
        return self.__pricer(self, option, data)


def BlackScholesDelta(spot, t, strike, expiry, volatility, rate, dividend):
    tau = expiry - t
    d1 = (np.log(spot/strike) + (rate - dividend + 0.5 * volatility * \
          volatility) * tau) / (volatility * np.sqrt(tau))
    delta = np.exp(-dividend * tau) * norm.cdf(d1)
    return delta

def NaiveMonteCarloPricer(engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, vol, div) = data.get_data()
    replications = engine.replications
    dt = expiry / engine.time_steps
    disc = np.exp(-rate * dt)

    z = np.random.normal(size = replications)
```

```
    spotT = spot * np.exp((rate - div - 0.5 * vol * vol) * dt + vol * \
                          np.sqrt(dt) * z)
    payoffT = option.payoff(spotT)

    prc = payoffT.mean() * disc
    se = payoffT.std(ddof=1) / np.sqrt(replications)

    return (prc, se)

def PathwiseNaiveMonteCarloPricer(engine, option, data):
    ## You gotta put the code here!
    ## See my AssetPaths function from class
    pass

def AntitheticMonteCarloPricer(engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, vol, div) = data.get_data()
    replications = engine.replications
    dt = expiry / engine.time_steps
    disc = np.exp(-(rate - div) * dt)

    z1 = np.random.normal(size = replications)
    z2 = -z1
    z = np.concatenate((z1,z2))
    spotT = spot * np.exp((rate - div) * dt + vol * np.sqrt(dt) * z)
    payoffT = option.payoff(spotT)

    prc = payoffT.mean() * disc

    return prc


def ControlVariatePricer(engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, volatility, dividend) = data.get_data()
    dt = expiry / engine.time_steps
    nudt = (rate - dividend - 0.5 * volatility * volatility) * dt
    sigsdt = volatility * np.sqrt(dt)
    erddt = np.exp((rate - dividend) * dt)
    beta = -1.0
    cash_flow_t = np.zeros((engine.replications, ))
    price = 0.0
```

```
    for j in range(engine.replications):
        spot_t = spot
        convar = 0.0
        z = np.random.normal(size=int(engine.time_steps))

        for i in range(int(engine.time_steps)):
            t = i * dt
            delta = BlackScholesDelta(spot, t, strike, expiry, volatility,
                                      rate, dividend)
            spot_tn = spot_t * np.exp(nudt + sigsdt * z[i])
            convar = convar + delta * (spot_tn - spot_t * erddt)
            spot_t = spot_tn

        cash_flow_t[j] = option.payoff(spot_t) + beta * convar

    price = np.exp(-rate * expiry) * cash_flow_t.mean()
    #stderr = cash_flow_t.std() / np.sqrt(engine.replications)
    return price

#class BlackScholesPayoffType(enum.Enum):
#    call = 1
#    put = 2

class BlackScholesPricingEngine(PricingEngine):
    def __init__(self, payoff_type, pricer):
        self.__payoff_type = payoff_type
        self.__pricer = pricer

    @property
    def payoff_type(self):
        return self.__payoff_type

    def calculate(self, option, data):
        return self.__pricer(self, option, data)

def BlackScholesPricer(pricing_engine, option, data):
    strike = option.strike
    expiry = option.expiry
    (spot, rate, volatility, dividend) = data.get_data()
    d1 = (np.log(spot/strike) + (rate - dividend + 0.5 * volatility * \
          volatility) * expiry) / (volatility * np.sqrt(expiry))
    d2 = d1 - volatility * np.sqrt(expiry)
```

```
if pricing_engine.payoff_type == "call":
    price = (spot * np.exp(-dividend * expiry) * norm.cdf(d1)) - \
    (strike * np.exp(-rate * expiry) * norm.cdf(d2))
elif pricing_engine.payoff_type == "put":
    price = (strike * np.exp(-rate * expiry) * norm.cdf(-d2)) - \
    (spot * np.exp(-dividend * expiry) * norm.cdf(-d1))
else:
    raise ValueError("You must pass either a call or a put option.")

#try:
#    #if pricing_engine.payoff_type == BlackScholesPayoffType.call:
#    if pricing_engine.payoff_type == "call":
#        price = (spot * np.exp(-dividend * expiry) * norm.cdf(d1)) - \
#(strike * np.exp(-rate * expiry) * norm.cdf(d2))
#    #else pricing_engine.payoff_type == BlackScholesPayoffType.put:
#    else pricing_engine.payoff_type == "put":
#        price = (strike * np.exp(-rate * expiry) * norm.cdf(-d2)) - \
#(spot * np.exp(-dividend * expiry) * norm.cdf(-d1))
#except ValueError:
#    print("You must supply either a call or a put option to the \
#BlackScholes pricing engine!")

return price
```