```python
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Callable

import numpy as np
import pandas as pd
from numpy.polynomial import Polynomial
from scipy.integrate import quad
from scipy.stats import norm


@dataclass
class Option:
    """
    Representation of an option derivative
    """

    s0: float
    T: int
    K: int
    v0: float = None
    call: bool = True

    def payoff(self, s: np.ndarray) -> np.ndarray:
        payoff = np.maximum(s - self.K, 0) if self.call else np.maximum(self.K - s, 0)
        return payoff


class StochasticProcess(ABC):
    """Represente a stocastic process (just for typing)"""

    @abstractmethod
    def simulate(self):
        ...


@dataclass
class GeometricBrownianMotion(StochasticProcess):
    """
    A classic geometric brownian motion which can be simulated.
    The closed form formula allow a fully vectorized calculation of the paths.
    """

    mu: float
    sigma: float

    def simulate(
        self,
        s0: float,
        T: int,
        n: int,
        m: int,
        v0: float = None,
        antithetic: bool = False,
        seed: int = None,
    ) -> pd.DataFrame:  # n = number of path, m = number of discretization points

        if seed:
            np.random.seed(seed)

        signe = -1 if antithetic else 1

        dt = T / m
        W = signe * np.cumsum(np.sqrt(dt) * np.random.randn(m + 1, n), axis=0)
        W[0] = 0

        T = np.ones(n).reshape(1, -1) * np.linspace(0, T, m + 1).reshape(-1, 1)

        s = s0 * np.exp((self.mu - 0.5 * self.sigma ** 2) * T + self.sigma * W)
```

```python
        return s


@dataclass
class HestonProcess(StochasticProcess):
    """
    An Heston process which can be simulated using Milstein schema.
    """

    mu: float
    kappa: float
    theta: float
    eta: float
    rho: float
    milstein: bool = True

    def simulate(
        self,
        s0: float,
        v0: float,
        T: int,
        n: int,
        m: int,
        return_vol: bool = False,
        antithetic: bool = False,
        seed: int = None,
    ) -> pd.DataFrame:  # n = number of path, m = number of discretization points

        if seed:
            np.random.seed(seed)

        dt = T / m
        z1 = np.random.randn(m, n)
        z2 = self.rho * z1 + np.sqrt(1 - self.rho ** 2) * np.random.randn(m, n)

        signe = -1 if antithetic else 1
        z1, z2 = z1 * signe, z2 * signe

        s = np.zeros((m + 1, n))
        x = np.zeros((m + 1, n))
        v = np.zeros((m + 1, n))

        s[0] = s0
        v[0] = v0

        for i in range(m):

            v[i + 1] = (
                v[i]
                + self.kappa * (self.theta - v[i]) * dt
                + self.eta * np.sqrt(v[i] * dt) * z1[i]
                + (self.eta ** 2 / 4 * (z1[i] ** 2 - 1) * dt if self.milstein else 0)
            )
            v = np.where(v > 0, v, -v)

            x[i + 1] = x[i] + (self.mu - v[i] / 2) * dt + np.sqrt(v[i] * dt) * z2[i]

            s[1:] = s[0] * np.exp(x[1:])

        return s if not return_vol else (s, v)


def monte_carlo_simulation(
    option: Option,
    process: StochasticProcess,
    n: int,
    m: int,
    alpha: float = 0.05,
```

```python
    return_all: bool = False,
    antithetic: bool = False,
    seed: int = None,
) -> float:
    """
    Given an option and a process followed by the underlying,
    calculate the classic monte carlo price estimator
    """
    # n = number of path, m = number of discretization points
    s = process.simulate(
        s0=option.s0,
        v0=option.v0,
        T=option.T,
        n=n,
        m=m,
        antithetic=antithetic,
        seed=seed,
    )
    st = s[-1]
    payoffs = option.payoff(s=st)

    discount = np.exp(-process.mu * option.T)
    price = np.mean(payoffs) * discount

    quantile = norm.ppf(1 - alpha / 2)
    confidence_interval = [
        np.round(price - quantile * np.std(payoffs * discount) / np.sqrt(n), 2),
        np.round(price + quantile * np.std(payoffs * discount) / np.sqrt(n), 2),
    ]

    print(f"The price of {option!r} = {price:.2f}")
    print(f"{(1-alpha)*100}% confidence interval = {confidence_interval}")

    return (
        np.round(price, 3)
        if not return_all
        else (np.round(price, 3), payoffs * discount)
    )


def monte_carlo_simulation_LS(
    option: Option,
    process: StochasticProcess,
    n: int,
    m: int,
    return_all: bool = False,
    antithetic: bool = False,
    seed: int = None,
) -> float:
    """
    Given an option and a process followed by the underlying,
    calculate the option value using the Longstaff-Schwartz algorithme
    """
    # n = number of path, m = number of discretization points

    s = process.simulate(
        s0=option.s0,
        v0=option.v0,
        T=option.T,
        n=n,
        m=m,
        antithetic=antithetic,
        seed=seed,
    )

    payoffs = option.payoff(s=s)

    v = np.zeros_like(payoffs)
    v[-1] = payoffs[-1]
```

```python
    dt = option.T / m
    discount = np.exp(-process.mu * dt)

    for t in range(m - 1, 0, -1):
        polynome = Polynomial.fit(s[t], discount * v[t + 1], 5)
        c = polynome(s[t])
        v[t] = np.where(payoffs[t] > c, payoffs[t], discount * v[t + 1])

    price = discount * np.mean(v[1])

    print(f"The price of {option!r} = {round(price, 4)}")

    return round(price, 3) if not return_all else (round(price, 3), v[1] * discount)


def black_scholes_merton(r, sigma, option: Option):
    """
    Calculate the price of vanilla options using BSM formula
    """
    d1 = (np.log(option.s0 / option.K) + (r + sigma ** 2 / 2) * option.T) / (
        sigma * np.sqrt(option.T)
    )
    d2 = d1 - sigma * np.sqrt(option.T)

    price = option.s0 * norm.cdf(d1) - option.K * np.exp(-r * option.T) * norm.cdf(d2)
    price = (
        price if option.call else price - option.s0 + option.K * np.exp(-r * option.T)
    )

    return np.round(price, 3)


def crr_pricing(
    r=0.1, sigma=0.2, option: Option = Option(s0=100, T=1, K=100, call=False), n=25000
):
    """
    Calculate the price of an americain option using a backward Cox,
    Ross and Rubinstein tree model
    """
    dt = option.T / n
    u = np.exp(sigma * np.sqrt(dt))
    d = 1 / u
    a = np.exp(r * dt)
    p = (a - d) / (u - d)
    q = 1 - p

    st = np.array([option.s0 * u ** i * d ** (n - i) for i in range(n + 1)])
    v = np.maximum(option.K - st, 0)

    for _ in range(n):
        v[:-1] = np.exp(-r * dt) * (p * v[1:] + q * v[:-1])
        st = st * u
        v = np.maximum(v, option.K - st)

    return np.round(v[0], 3)


def heston_semi_closed(option: Option, process: StochasticProcess):
    def caracteristic_function(
        j: int,
        phi: float,
        t: float,
        v0: float,
        mu: float,
        kappa: float,
        theta: float,
        eta: float,
        rho: float,
```

```python
) -> float:

    if j == 1:
        u = 1 / 2
        b = kappa - rho * eta
    else:
        u = -1 / 2
        b = kappa

    a = kappa * theta
    M = b - rho * eta * phi * 1j
    d = np.sqrt(M ** 2 - eta ** 2 * (2 * u * phi * 1j - phi ** 2))
    g = (M + d) / (M - d)

    C = mu * phi * 1j * t + (a / eta ** 2) * (
        (M + d) * t - 2 * np.log((1 - g * np.exp(d * t)) / (1 - g))
    )
    D = (M + d) / eta ** 2 * (1 - np.exp(d * t)) / (1 - g * np.exp(d * t))

    cf = np.exp(C + D * v0)

    return cf

def P(cf: Callable, K: float, j: int) -> float:

    func = lambda phi: np.real(
        np.exp(-phi * np.log(K) * 1j) / (phi * 1j) * cf(j=j, phi=phi)
    )

    return 1 / 2 + 1 / np.pi * quad(func, 1e-10, 1000, limit=1000)[0]

cf = lambda j, phi: caracteristic_function(
    j=j,
    phi=phi,
    t=option.T,
    v0=option.v0,
    mu=process.mu,
    theta=process.theta,
    eta=process.eta,
    kappa=process.kappa,
    rho=process.rho,
)

Pj = lambda j: P(cf=cf, K=option.K / option.s0, j=j)

price = option.s0 * Pj(j=1) - option.K * np.exp(-process.mu * option.T) * Pj(j=2)

if not option.call:
    price = price - option.s0 + option.K * np.exp(-process.mu * option.T)

return np.round(price, 3)
```