```python
'''
// TestMC101.py
//
// Prototype Monte Carlo option pricing in Python
//
// (C) Datasim Education BV 2008-2023
//
This code can be used as a baseline for extensions in C++ and C#, for example. More
generally,
other possible use cases are:

U1: Learning a language feature in Python as a stepping-stone to learning this feature in
C++ (quicker
turnaround time in Python). An example is the code below.
U2: Write Python prototype application and use it as input requirements to a C++
production version.
U3: Generalise an application using design patterns; pattern mining in code. Probably
easiest in Python.
U4: Write two applications, one in Python, the other in C++. Compare the solutions.
U5: Python (get it working) -> C++ (get it right) -> C++ with domain architectures and
design patterns.
U6: Hybrid C++/Python application.
..
U99: use all resources to solve problem.

“It is better to solve one problem five different ways, than to solve five problems one
way.”
— George Pólya


Book: Modern Multiparadigm Software Architectures and Design Patterns
with Examples and Applications in C++, C# and Python Volume I
Datasim Press (planned publication date December 2023)

Daniel J. Duffy dduffy@datasim.nl and Harold Kasperink harold.kasperink@enjoytocode.com

ISO/IEC 1926/25010 (top-level) software characteristics, for example:

Functionality
Maintainability
Portability
Efficiency
Usability
Reliability

In tests,we get a speedup of ~ 4 with 4 processors.

'''

# This Python code is a port from C++
import time
import numpy, math, cmath, random

from numpy.random import Generator, Philox, PCG64, MT19937,PCG64DXSM
import abc
from abc import ABC

from concurrent.futures import ProcessPoolExecutor
```

```python
import multiprocessing as mp
from concurrent import futures
#import threading

class Rng(ABC):
    @property
    @abc.abstractmethod
    def value(self):
        pass

class PhiloxRng(Rng):
    def __init__(self, seed = 1234):
         self.rg = Generator(Philox(seed))

    def value(self):
        return self.rg.standard_normal()

    def __call__(self):
        return self.value()

class PCG64Rng(Rng):
    def __init__(self, seed = 1234):
         self.rg = Generator(PCG64(seed))

    def value(self):
        return self.rg.standard_normal()

    def __call__(self):
        return self.value()

class PCG64DXSMRng(Rng):
    def __init__(self, seed = 1234):
         self.rg = Generator(PCG64DXSM(seed))

    def value(self):
        return self.rg.standard_normal()

    def __call__(self):
        return self.value()
class MT19937Rng(Rng):
    def __init__(self, seed = 1234):
         self.rg = Generator(MT19937(seed))

    def value(self):
        return self.rg.standard_normal()

    def __call__(self):
        return self.value()

class GaussRng(Rng):
    def value(self):
        return random.gauss(0,1)

    def __call__(self):
        return self.value()

# Put payoff
def Payoff(x, K):
```

```python
        return max(K -x,0.0)

t0 = time.time()

# Initialise the option data
r = 0.08
d = 0.0
sig = 0.3
T = 0.25
K = 65.0
S_0 = 60


# Exact
import numpy as np
from scipy.stats import norm

N = norm.cdf

def CallPrice(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * N(d1) - K * np.exp(-r*T)* N(d2)

#print ('Exact Call:  ', CallPrice(S_0, K, T, r, sig))

def PutPrice(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma* np.sqrt(T)
    return K*np.exp(-r*T)*N(-d2) - S*N(-d1)
#print ('Exact Put:  ', PutPrice(S_0, K, T, r, sig))

VOld = S_0

# NT = time steps, NSIM number of simulations
NT = 300; NSIM = 1000000
# discrete parameters
dt = T /NT;
sqrk = math.sqrt(dt)

def computePrice(NSIM, NT, rg):
    sumPriceT = 0.0
    for i in range (1,NSIM):
        VOld = S_0
        for j in range (0,NT):
            VNew = VOld + (dt*(r-d)*VOld) + (sqrk * sig*VOld * rg())
            VOld = VNew
        sumPriceT += Payoff(VNew, K)
    return sumPriceT

# Family of random number generators
rg1 = PhiloxRng()
rg2 = PCG64Rng()
rg3 = GaussRng()
rg4 = MT19937Rng()
```

```python
#
https://numpy.org/doc/stable/reference/random/bit_generators/pcg64dxsm.html#numpy.random.
PCG64DXSM
rg5 = PCG64DXSMRng()

'''
print("Sequential MC")
price = math.exp(-r * T) * computePrice(NSIM, NT,rg5) / NSIM
print(price)

price = math.exp(-r * T) * computePrice(NSIM, NT,rg4) / NSIM
print(price)

price = math.exp(-r * T) * computePrice(NSIM, NT,rg3) / NSIM
print(price)

price = math.exp(-r * T) * computePrice(NSIM, NT,rg2) / NSIM
print(price)

price = math.exp(-r * T) * computePrice(NSIM, NT,rg1) / NSIM
print(price)


t1 = time.time()
print("time to compute ", t1-t0)

'''

# Parallel version
def computePrice2 (NSIM, NT, rg):
    return math.exp(-r * T) * computePrice(NSIM, NT,rg) / NSIM


if __name__ == '__main__':
    print ('Exact Put:  ', PutPrice(S_0, K, T, r, sig))
    print ('Exact Call: ', CallPrice(S_0, K, T, r, sig))
    print ('Estimated processing time: [300,500] seconds, depending on NS and NT..')
    t0 = time.time()
    mp.set_start_method('spawn')

    with ProcessPoolExecutor(max_workers = 5) as pool:
        fut1 = pool.submit(computePrice2, NSIM, NT, rg1)
        fut2 = pool.submit(computePrice2, NSIM, NT, rg2)
        fut3 = pool.submit(computePrice2, NSIM, NT, rg3)
        fut4 = pool.submit(computePrice2, NSIM, NT, rg4)
        fut5 = pool.submit(computePrice2, NSIM, NT, rg5)

        print(fut1.result())
        print(fut2.result())
        print(fut3.result())
        print(fut4.result())
        print(fut5.result())

        t1 = time.time()
        print("time to compute in parallel ", t1-t0)
```