```python
### Cloning_randomtimes_tmax.py
### When cloning, the time is now drawn differently for each new copy.

import random, math, pylab, heapq
                # heapq provides the Heap Queue structure which allows to manipulate efficiently sorted list.
                # Here we use a list of copies of the system, sorted according to their next time of evolution.
c=.25           # c = creation rate 0->1 ; 1-c = annihilation rate 1->0
s=-0.02         # s conjugated to the activity K ; this algorithm works for s<0
populat=[(0.,.1,0)]
                # initial state of the population :
                # each member of (or "copy" in) the population is described by a pair (time,dt,state)
                #   time  = next time at which it will evolve
                #   dt = time since last evolution
                #   state = 0 or 1 = empty or occupied
t,tmax=0.,1300. # initial and maximal time
step=0          # counter for the number of steps in the "mutation/selection" process

# s-modified escape rate
def sescape(n):
    if n==0:
        esc=c*math.exp(-s)
    else:
        esc=(1-c)*math.exp(-s)
    return esc

def cloningrate(n):  # always positive, as we took s<0
    if n==0:
        rate=c*(math.exp(-s)-1.)
    else:
        rate=(1.-c)*(math.exp(-s)-1.)
    return rate

# lists to sample the logarithm of the population size a function of time
samplestime,samplespop=[],[]

heapq.heapify(populat)  # orders the population into a Heap Queue ; useful if the initial state of the population contains more than one copy
while t<tmax:
    (t,dt,state)=heapq.heappop(populat) # pops and yields the first element of populat, which is always the next to evolve
    p=int( math.exp(dt*cloningrate(state)) + random.random() ) # the copy we poped out is to be replaced by p copies
    while p>0:
        p-=1
        Deltat=random.expovariate(sescape(1-state)) # interval until next evolution
        toclone=(t+Deltat,Deltat,1-state)
        heapq.heappush(populat,toclone)
    step+=1
    if step%10 == 0:
        samplestime.append(t)
        samplespop.append(math.log(len(populat)))

# Bulk numerical estimation of psiK(s) from population size ~ e^(t psiK(s) )
psiK=math.log(len(populat))/t

# better estimation by fitting log(popsize(t)) starting from a given threshold so as to isolate the large-time exponential behaviour popsize(t) ~ e^(t psiK(s) )
Nsamples=len(samplestime)
threshold=Nsamples/2
psiKfit,const = pylab.polyfit(samplestime[threshold:],samplespop[threshold:],1)

print 'final total population size = ', len(populat)
print '         theoretical psi(s) = ', -.5+.5*math.sqrt(1.-4.*c*(1.-c)*(1.-math.exp(-2.*s)))
print '      bulk numerical psi(s) = ', psiK
print 'fitted numerical fit psi(s) = ', psiKfit

pylab.plot(samplestime,samplespop, 'r')
pylab.plot(samplestime,[const+psiKfit*samplestime[i] for i in range(Nsamples) ], 'g-')
pylab.show()
```