Assignment 10 1 - Program (Loop)

```
#use both numpy (for maths and random number generator); and sys (for command line arguments)
import numpy
import sys

#for this program, the number of samples to use is read from the command line
if (len(sys.argv) < 2):
  print 'Please supply command line argument indicating the number of sample points to use:'
  print 'python assign_10_1.py [number]'
  sys.exit()

#tot_points = int(raw_input('Please specify the number of sample points: ')) #alternative form for reading interactively
tot_points = int(sys.argv[1]) #use the argument at index ‘1’ (note that these are arguments relative to the ‘python’ call itself; ie sys.argv[0] is the name of the script being run)
circ_points = 0 #initialise circle count

for i in range(tot_points):
  x,y = numpy.random.rand(2) #numpy.random.rand() is a simple call to the uniform random number generator; the ‘2’ argument indicates to return two values
  if (x*x + y*y < 1.0):
    #perform calculation to check if inside circle, and increment counter if it is
    circ_points += 1

#calculation for pi
sim_pi = 4.0 * float(circ_points)/float(tot_points)

#print to screen simulated value, true value and percentage difference
print '%d iterations'%(tot_points)
print 'Simulated Pi: ', sim_pi
print 'True Pi      : ', numpy.pi
print 'Percentage Difference: ', numpy.abs((sim_pi - numpy.pi)/(numpy.pi)) * 100., '%'
```

Assignment 10 1 - assign_10_1.out

```
1000 iterations
Simulated Pi:  3.144
True Pi      :  3.14159265359
Percentage Difference:  0.0766282161838 %

user  0m0.078s

10000 iterations
Simulated Pi:  3.1316
True Pi      :  3.14159265359
Percentage Difference:  0.318076042684 %

user  0m0.125s

100000 iterations
Simulated Pi:  3.13856
True Pi      :  3.14159265359
Percentage Difference:  0.0965323619002 %
```

```

user   0m0.622s

1000000 iterations
Simulated Pi:   3.141832
True Pi      :   3.14159265359
Percentage Difference:   0.00761863285914 %

user   0m5.853s



Note in the above file how long these calculations have taken; compare with your own Fortran program; where even 1000000 iterations should have taken no more than a few tens of milliseconds. Also compare with the output for 'assign_10_1_alternate.out'; where a 'vectorised' array operation set was used instead of a loop.
```

Assignment 10 1 - Program (Vectorised)

```python
#This is an alternative version of the other assign_10_1.py program. In this, rather than using a loop, as we might in Fortran (but which is very slow in python), we use an array operation. Note that this can be done in Fortran too, but evaluating it for the logicals is not done with as much ease.

#use both numpy (for maths and random number generator); and sys (for command line arguments)
import numpy
import sys

#for this program, the number of samples to use is read from the command line
if (len(sys.argv) < 2):
  print 'Please supply command line argument indicating the number of sample points to use:'
  print 'python assign_10_1.py [number]'
  sys.exit()

#tot_points = int(raw_input('Please specify the number of sample points: ')) #alternative form for reading interactively
tot_points = int(sys.argv[1]) #use the argument at index '1' (note that these are arguments relative to the 'python' call itself; ie sys.argv[0] is the name of the script being run)
circ_points = 0 #initialise circle count

xs = numpy.random.rand(tot_points) #get *all* of the x coordinates in one go
ys = numpy.random.rand(tot_points) #get *all* of the y coordinates in one go
check = (xs*xs + ys*ys < 1.0) #new array of *all* checks in one go; this is called a 'vectorised' operation, where a very fast form of loop can go through all of the elements, as *exactly* what is required by the entire loop is known in advance
circ_points = numpy.sum(check) #'sum' of a logical array is +1 for every 'True' value, +0 for every 'False' value

#calculation for pi
sim_pi = 4.0 * float(circ_points)/float(tot_points)

#print to screen simulated value, true value and percentage difference
print '%d iterations'%(tot_points)
print 'Simulated Pi: ', sim_pi
```

```
print 'True Pi        : ', numpy.pi
print 'Percentage Difference: ', numpy.abs((sim_pi - numpy.pi)/(numpy.pi)) * 100., '%'
```

Assignment 10 1 - assign_10_1_alternate.out

```
1000 iterations
Simulated Pi:  3.088
True Pi      :  3.14159265359
Percentage Difference:  1.70590714645 %

user  0m0.081s

10000 iterations
Simulated Pi:  3.13
True Pi      :  3.14159265359
Percentage Difference:  0.369005624474 %

user  0m0.080s

100000 iterations
Simulated Pi:  3.13508
True Pi      :  3.14159265359
Percentage Difference:  0.207304202292 %

user  0m0.083s

1000000 iterations
Simulated Pi:  3.141776
True Pi      :  3.14159265359
Percentage Difference:  0.00583609749652 %

user  0m0.135s


Compare these times with the 'assign_10_1.out' output. Far faster, but still not as fast as even the regular-loop format Fortran codes.
```

Assignment 10 2 - Program

```
#read in all desired numbers
a = float(raw_input('Please specify the number to take the square root of: '))
x = float(raw_input('Please specify the initial guess: '))
n = int(raw_input('Please specify the number of iterations: '))

#prepare output file
fout = open('assign_10_2.out','w')

#display initial conditions in file
fout.write('Attempting to find square root of %f, using initial guess of %f, over %d iterations.\n'%(a,x,n))

for i in range(n):
  x = x - (x**2 - a)/(2.0 * x) #newton-raphson calculation for new x
  fout.write('%4d %f\n'%(i+1,x)) #output new estimate of solution to file
  print i+1, x #..and to screen

```

```
fout.close() #close file
```

Assignment 10 2 - Output

```
Attempting to find square root of 131.280000, using initial guess of
   200.000000, over 10 iterations.
    1 100.328200
    2 50.818353
    3 26.700836
    4 15.808768
    5 12.056510
    6 11.472617
    7 11.457758
    8 11.457748
    9 11.457748
   10 11.457748
```