Writing A Program

February 21, 2019

1 Writing a program to compute π

1.1 Getting random

For a Monte Carlo method, we need a random number generator.

```
In [1]: import random
        random.random()
```

```
Out[1]: 0.13636199596966703
```

Enter the following lines into Spyder's editor and run the file.

```
In [2]: import random
        print(random.random())
```

```
0.6542716249221561
```

When you're running a program, you have to tell Python when to `print`.
We'll need pairs of random numbers. We can put them into tuples. Try this program:

```
In [3]: import random
        a = (random.random(), random.random())
        print(a)
        print(a[0])
        print(a[1])
```

```
(0.8906722613370666, 0.8624023536198429)
0.8906722613370666
0.8624023536198429
```

Actually, we're going to need *lots* of random tuples...
...so let's rewrite the program

1.1.1 Assignment #1

1. Import the `random` module
2. Define an empty list: `L = []`
3. Start a `for` loop with `range(10000)`
4. In the loop, use `L.append()` to add random tuples to `L`
5. After the loop, print the last member of `L`
6. *Extra credit:* print the length of `L`

1.1.2 Answer to assignment #1

```
In [4]: import random
        L = []
        for i in range(10000):
          L.append((random.random(), random.random()))
        print(L[-1])
        print(len(L))
```

```
(0.9141208455640527, 0.33665846842681857)
10000
```

1.1.3 Assignment #2

- Click on `random()` in Spyder's editor and type Alt-i or Cmd-i
  - The help pane contains the same info as `help(random.random)`
  - What kind of random numbers are we getting?
- We want uniform random numbers in the interval (-1,+1)
- **Modify the body of the loop to multiply and shift the random numbers**

1.1.4 Answer to assignment #2

```
In [5]: import random
        L = []
        for i in range(10000):
          L.append((2*random.random()-1, 2*random.random()-1))
        print(L[-1])
        print(len(L))
```

```
(-0.3588265620779476, -0.8266254895591438)
10000
```

1.2 Functions

We are going to split our program into a function followed by a main program.

1.2.1 Why functions?

- They give organization and structure to your code
- They can be tested and verified separately ("unit testing")
- They help you build your program incrementally

- *Alternative:* define a class with methods (also good, but more complicated)

1.2.2 What should go in a function?

- A group of statements that achieves a single goal or result
- The goal should be identifiable and separable (easy to name!)
- The goal should also be natural and logical (not too trivial or too huge)

- *Note:* A function may call other functions to do sub-tasks.

1.2.3 Assignment #3

Try it! Use this template to split your program into a function and a main program.

In [6]:
```
import random

def make_list_of_random_coords():
    L = []
    # loop to populate list
    return L

if __name__ == '__main__':
    darts = make_list_of_random_coords()
    print(type(darts), len(darts))
```

```
<class 'list'> 0
```

1.2.4 Assignment #4

Add a parameter `n` to the function, and call it with an argument as shown.

In [7]:
```
import random

def make_list_of_random_coords(n):
    L = []
    # loop with range(n) to populate list
    return L

if __name__ == '__main__':
    darts = make_list_of_random_coords(10000)
    print(type(darts), len(darts))
```

```
<class 'list'> 0
```

1.2.5 Answer to assignments #3 and #4

```
In [8]: import random

        def make_list_of_random_coords(n):
            L = []
            for i in range(n):
                L.append((2*random.random()-1, 2*random.random()-1))
            return L

        if __name__ == '__main__':
            darts = make_list_of_random_coords(10000)
            print(type(darts), len(darts))
```

```
<class 'list'> 10000
```

1.2.6 Adding a function

We need a second function to tell us which coordinate pairs fall inside the unit circle.

1.2.7 What kind of function do we need?

- Input is our list of random samples
- Output could be just a count of how many are in the unit circle…
- But we'll want to know which points are in or out, so we can plot them later

1.2.8 Assignment #5

Define a second function, `unit_circle_check(L)`, which should: 1. Create an empty list U 2. Loop over coordinate pairs P in input list L: - Compute the distance from (0,0) to P - Test if the distance is is less than 1 - Append the result of the test to U - See if you can do all that in one line 3. Return U

1.2.9 Answer to assignment #5

```
In [9]: def unit_circle_check(L):
            U = []
            for P in L:
                U.append(((P[0]*P[0] + P[1]*P[1])**0.5)<1)
            return U
```

1.3 Testing

It's an essential part of code development.

1.3.1 How do we know if our new function works?

- *We don't!* The only way to tell is to test
- *Expect errors!* Diagnose, fix, test again
- We can add temporary code to `__main__` to run the test
- In a production code, this would be saved as a **unit test**

1.3.2 Assignment #6

At the end of the main program, add code that: 1. Creates a short list of fixed coordinate pairs 2. Calls `unit_circle_check` with the short list 3. Prints the return values for inspection

Suggestion: Try removing the `< 1` from the function so it just returns distances

1.3.3 Answer to assignment #6

```
In [10]: # previous function can stay here

         def unit_circle_check(L):
             U = []
             for P in L:
                 U.append(((P[0]*P[0] + P[1]*P[1])**0.5)<1)
             return U

         if __name__ == '__main__':
             # previous code can stay here

             few_darts = [(0.3,0.4),(-1,-1)]
             hits = unit_circle_check(few_darts)
             print(hits)
```

```
[True, False]
```

1.3.4 At this point, we have what we need to compute π. Can we do it?

Put this in `__main__` to find out:

```
In [11]: if __name__ == '__main__':
             darts = make_list_of_random_coords(10000)
             hits = unit_circle_check(darts)
             pi_approx = 4 * hits.count(True) / len(hits)
             print("pi_approx = ", pi_approx)
```

```
pi_approx =  3.1608
```

Not a great approximation. Did we do something wrong?

1.3.5 More testing!!

- Quick tests are good for fixing obvious errors
- However, it's also good to do extensive tests for many inputs
- We can't look at all this output as text, but we can plot results...

Visualization and testing often go together!

1.4 Visualizing output

We can use matplotlib to make a scatter plot, after rearranging the output with NumPy.

```
In [12]: # this is just for me
         %matplotlib inline
```

1.4.1 Numpy has useful tools for wrangling data

- `numpy.array()` lets you create a NumPy array from array-like data types
- `numpy.where()` lets you select data from an array according to a condition

This shows how to make a 2D array from a list of tuples, and then slice it into two 1D arrays.

```
In [13]: import numpy as np
         b = np.array([(1,2),(3,4),(5,6)])
         x = b[:,0]
         y = b[:,1]
         print(x,y)
```

```
[1 3 5] [2 4 6]
```

Then we can select points in the x and y lists according to some condition.

```
In [14]: import numpy as np
         b = np.array([(1,2),(3,4),(5,6)])
         x = b[:,0]
         y = b[:,1]
         print(x,y)

         selected_points = np.where(x+y>3)
         print(x[selected_points],y[selected_points])
```

```
[1 3 5] [2 4 6]
[3 5] [4 6]
```

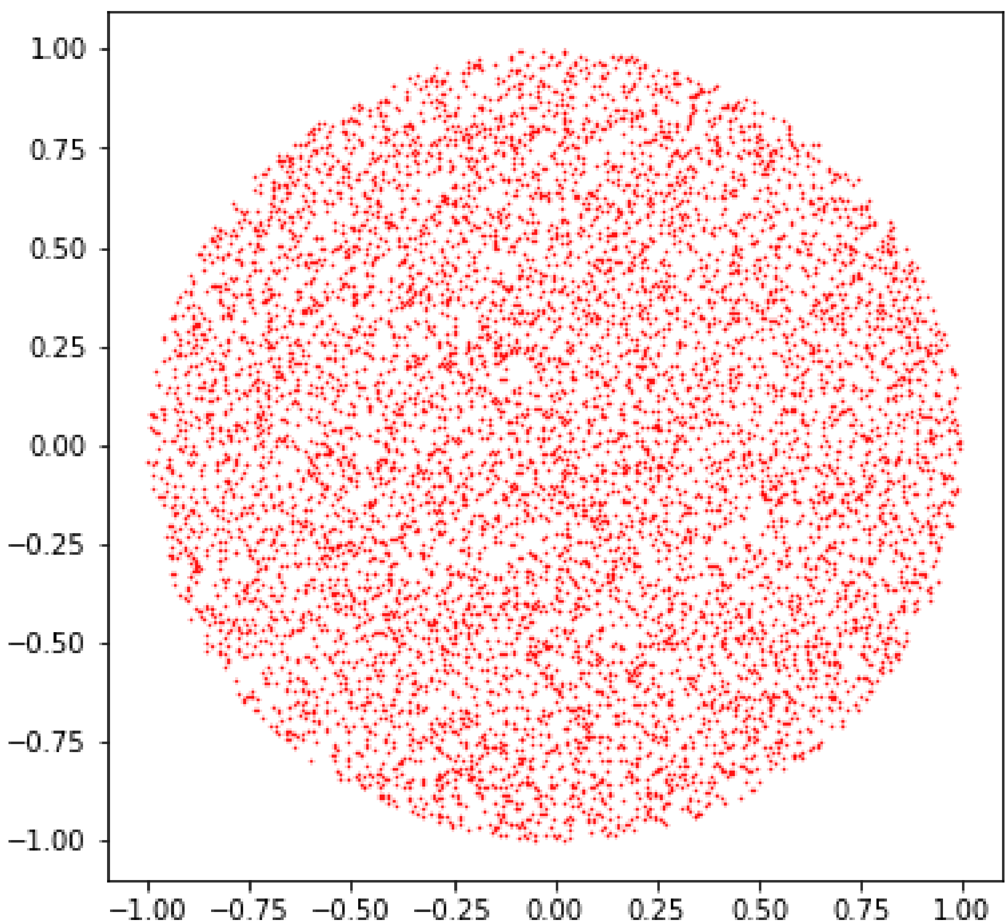
1.4.2 Assignment #7

Prepare for plotting by putting these tricks in the program. (Yes, just type it in.)

```
In [15]: import matplotlib.pyplot as plt
         import numpy as np
         import random
         # functions go here

         if __name__ == '__main__':
         # main program with pi computation goes here

             d_arr = np.array(darts)
             x = d_arr[:,0]
             y = d_arr[:,1]
             h_arr = np.array(hits)
```

And finally, make a plot!

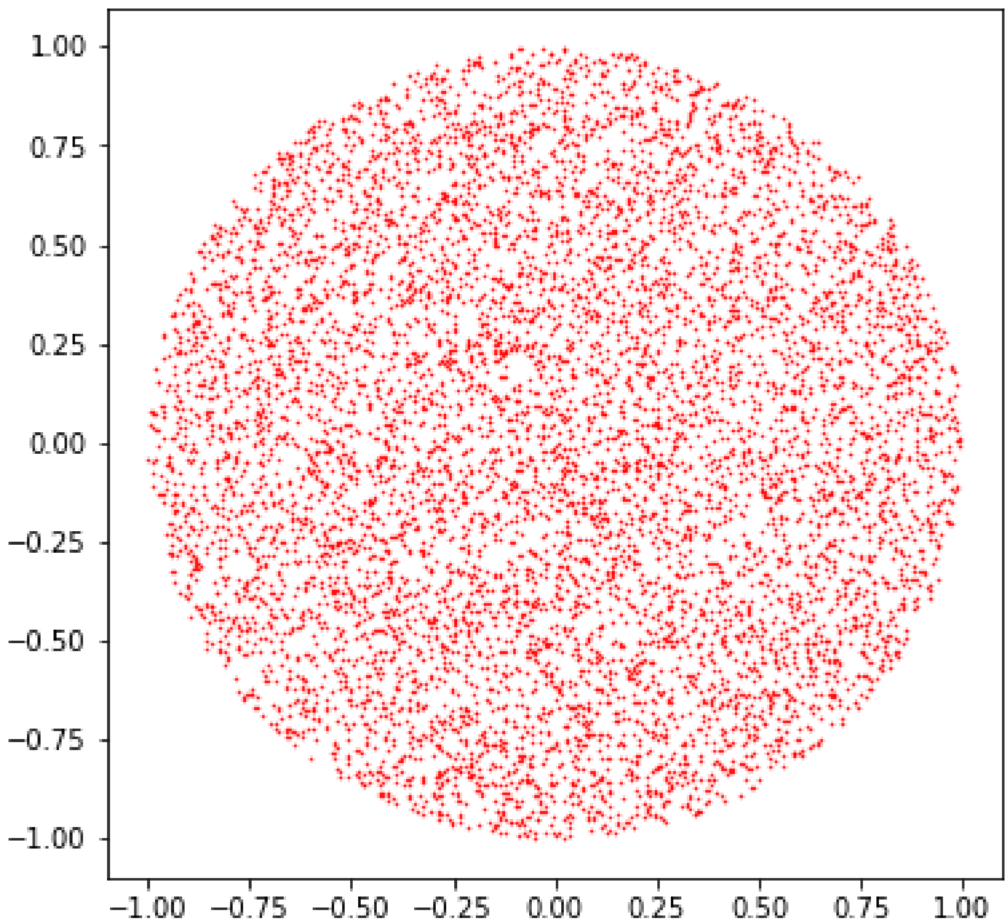
```
In [16]:     plt.rcParams['figure.figsize'] = (6,6)
             plt.scatter(x[np.where(h_arr)], y[np.where(h_arr)], s=1, c='r', marker='.')

Out[16]: <matplotlib.collections.PathCollection at 0x1146d5ba8>
```



1.4.3 Assignment #8

In [17]:
```python
plt.rcParams['figure.figsize'] = (6,6)
plt.scatter(x[np.where(h_arr)], y[np.where(h_arr)], s=1, c='r', marker='.')
# Call plt.scatter again to add points that are outside the unit circle, in blue.
# Note, in NumPy, you can use ~ (tilde) to negate an array of booleans
```

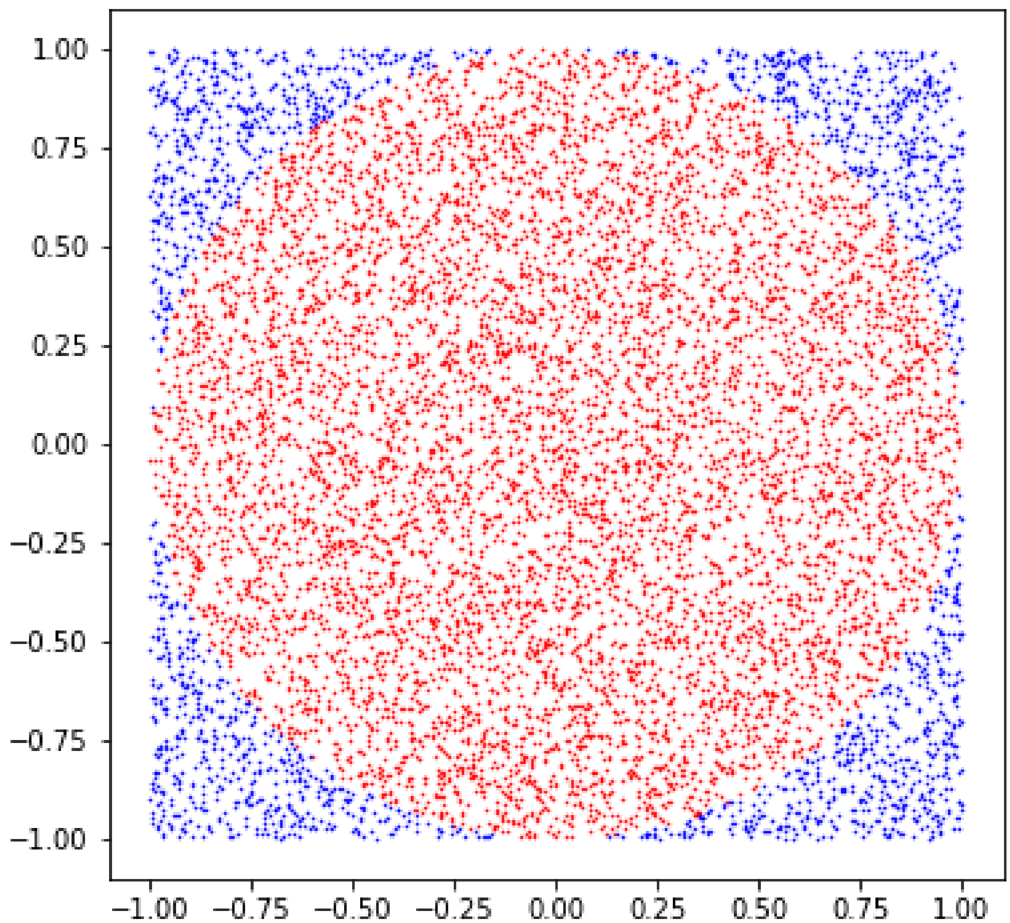
Out[17]: <matplotlib.collections.PathCollection at 0x114838f98>



1.4.4 Answer to assignment #8

In [18]:
```python
plt.rcParams['figure.figsize'] = (6,6)
plt.scatter(x[np.where(h_arr)], y[np.where(h_arr)], s=1, c='r', marker='.')
plt.scatter(x[np.where(~h_arr)], y[np.where(~h_arr)], s=1, c='b', marker='.')
```

Out[18]: <matplotlib.collections.PathCollection at 0x11487ba90>



The plot indicates that the coordinate pairs are valid, and `unit_circle_check` works correctly...

It looks like we just need a lot more samples to get a good approximation to π.

Maybe we should have been using NumPy from the beginning?