Python #11
Random Numbers, Simulation, Testing

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python11/python11.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Random Numbers

- *Random numbers are used for sampling and testing;*
- *Random numbers are computed by a random number generator (RNG);*
- *The random number generator can be initialized, and reinitialized, with a "seed";*
- *The* `numpy` *library provides a choice of RNG's;*
- *A* `numpy` *RNG can:*
  - *return integers or floats as scalars or arrays;*
  - *return a random permutation;*
  - *fill an array with repeated selections from a set;*
  - *sample normal, binomial, logistic, and other distributions;*
  - *estimate probabilities, integrals, or areas;*
  - *test a hypothesis on many random examples;*
  - *simulate a complicated process (forest fires, epidemics, nuclear reactors);*

It may seem surprising that random numbers play an important role in computing. After all, it seems that if we use random numbers, we will get random results, and so it will all be meaningless. But, as we will see, random numbers can be used as a flexible way of sampling complex sets, sequences, intervals, geometric regions, and various processes.

We might want to know the probability that a given bridge player has at most 10 High Card Points (Ace=4/King=3/Queen=2/Jack=1) after all 52 cards are dealt. The number of different ways a deck of cards can be dealt out is unimaginably large, so we can't write them all down and then count the number of times our player stays at 10 points or under. There may, of course, be a mathematical approach to this problem, but even if you can work it out, you might be unsure of your result. By using random numbers, we can generate a large number of examples to analyze quickly, and compare with any mathematical result we have come up with. Randomness makes it easy to do this generation, and also tends to create a well spread out set of examples that are representative of the much larger (in fact, enormous) set of total cases.

In previous discussions, we have often requested random numbers using calls like

```
import numpy as np
x = np.random ( 5, 3 )
```

However, new and flexible random number generation schemes have been added to `numpy`. These take a small bit of work to start up, but then provide a powerful tool for computation. Thus, at least for this discussion, we will `never` use the old style of random number computation.

1 Starting up a generator

To generate random numbers, we need to identify our random number generator. For our purposes, it will be enough to use the following statement, which creates an object `rng` which we will access for all our needs:

```
rng = np.random.default_rng ( )
```

We can then call functions such as `rng.integers()`:

```
r = rng.integers ( low = 0, high = 100, size = 10 )
```

If you rerun these commands or if a friend executes these same commands, a **different** set of random integers will be returned. Usually, this is fine.

However, if you are performing some kind of experiment which needs to be repeatable, you can guarantee that the random number generator will produce the exact same output on every run by specifying a **seed** when you define `rng`. In the following example, we define `rng` three times, with a seed. Because the first and third definitions use the same seed, we will get the same random values on those two occasions:

```
for seed in [ 123456789, 987654321, 123456789 ]:
  rng = np.random.default_rng ( seed )
  r = rng.random ( 5 )
  print ( r )
```

In a normal situation, we will just need to define `rng` once, and we can skip specifying the seed value unless we want repeatable results.

2 Random integers

The function `rng.integers()` will return random integers in a specified range. Although we specify a `high` value for the range, the actual values returned will never exceed `high-1`, in typical Python fashion. Consider the following examples of computing random integers:

```
r = rng.integers ( low = 0, high = 100 )
r = rng.integers ( low = 0, high = 100, size = 1 )
r = rng.integers ( low = 0, high = 100, size = 3 )
r = rng.integers ( low = 0, high = 100, size = ( 3, 2 ) )
```

3 Random floats

The function `rng.random()` returns random real numbers from the range $[0, 1]$.

```
r = rng.random ( )
r = rng.random ( 1 )
r = rng.random ( 3 )
r = rng.random ( 3, 2 )
```

The values returned by `rng.random()` follow the uniform distribution on [0,1]. This distribution has mean $\mu = 0.5$ and standard deviation $\sigma = \frac{1}{\sqrt{12}}$. A sequence of values generated by `rng.random()` should have approximately these statistics:

```
x = rng.random ( 10000 )
x_mean = np.mean ( x )
x_std = np.std ( x )
x_min = np.min ( x )
x_max = np.max ( x )
```

If you want your values to be uniformly spread over the interval $[a, b]$, do this:

```
a = np.pi
b = 10.0
r = a + ( b - a ) * rng.random ( )
r = a + ( b - a ) * rng.random ( 1 )
r = a + ( b - a ) * rng.random ( 3 )
r = a + ( b - a ) * rng.random ( 3, 2 )
```

4 Standard normal random values

The `rng.standard_normalm()` function returns real numbers that are normally distributed over the entire real line. These values are generated in such a way that their average is 0, and their standard deviation is 1. (This is why they are related to the **standard** normal distribution.) The normal distribution is sometimes called the "bell curve". Here is how we could get such samples:

```
print ( '' )
r = rng.standard_normal ( )
print ( '  r = rng.standard_normal ( )' )
print ( r )
r = rng.standard_normal ( 1 )
print ( '  rng.standard_normal ( 1 )' )
print ( r )
r = rng.standard_normal ( 3 )
print ( '  rng.standard_normal ( 3 )' )
print ( r )
r = rng.standard_normal ( ( 3, 2 ) )
print ( '  rng.standard_normal ( ( 3, 2 ) )' )
print ( r )
```

The values returned by `rng.standard_normal()` follow the normal distribution with mean $\mu = 0.0$ and standard deviation $\sigma = 1$. A sequence of values generated by `rng.standard_normal()` should have approximately these statistics:

```
x = rng.standard_normal ( 10000 )
x_mean = np.mean ( x )
x_std = np.std ( x )
x_min = np.min ( x )
x_max = np.max ( x )
```

If you want to specify the mean or standard deviation of the normal distribution you are using, you can call `rng.random`. Here, we request values with mean $\mu = 10$ and standard deviation $\sigma = 2.0$:

```
x = rng.normal ( 10.0, 2.0, 10000 )
```

We could, instead, have computed standard normal values and then transformed them:

```
x = rng.standard_normal ( )
x = 10.0 + 2.0 * x
```

5 Randomly permute a vector

```
print ( '' )
a = np.arange ( 10 )
print ( '  a = np.arange ( 10 )' )
print ( a )
b = rng.permutation ( a )
print ( '  b = rng.permutation ( a )' )
print ( b )
```

6 Randomly permute a matrix

To randomly permute a matrix, we want to randomly swap rows, and randomly swap columns. However, the `rng.permutation()` function will only swap items in the first dimension of an object. So it's easy to permute the rows of a matrix. If we also want to permute the columns, we have to transpose, swap, and then transpose back. This can be done in a sequence of function calls, or in one scary call:

```
c = np.array ( [ \
  [  0,  1,  2,  3,  4 ], \
  [ 10, 11, 12, 13, 14 ], \
  [ 20, 21, 22, 23, 24 ] ] )

d = rng.permutation ( c )       # permute the rows
e = np.transpose ( d )          # transpose
f = rng.permutation ( e )       # permute the columns
g = np.transpose ( f )          # transpose back

defg = np.transpose ( rng.permutation ( np.transpose ( rng.permutation ( c ) ) ) )
```

7 Random selection, no repeats

Suppose we have 12 items, and we want to choose 3 of them at random. One way to do this would be to set up an array of indices, 0 through 9, then randomly permute them, and use the first three:

```
months = np.array ( [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', \
                      'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec' ] )
i = np.arange ( 10 )
j = rng.permutation ( i )
three_months = months[j[0:3]]
```

If we are want to deal 5 cards from a shuffled deck, then we can identify each card with a number between 0 and 51, and so our process is even simpler:

```
deck = np.arange ( 52 )
shuffled_deck = rng.permutation ( deck )
hand = shuffled_deck[0:5]
```

8 Randomly selection, repeats allowed

Suppose that we want to pick five cards from a deck, but after each card is picked and observed, we return it to the deck. That means that it's quite possible that our list of picked cards will have repeats. In fact, if we pick more than 52 times, we are sure to see such repeats.

The function `rng.choice()` is given a set of options, and a list of how many times an option is to be chosen at random:

```
deck = np.arange ( 52 )
b = rng.choice ( deck, size = 5 )
```

Using the same function, we could create at random a win/loss/tie record for a team that will be playing 10 games in a season. Of course, here we will be assuming in advance that, for each game, the three outcomes are equally likely. If we do this twice, we get two different season records:

```
winlosstie = [ 'win', 'loss', 'tie' ]
record = rng.choice ( winlosstie, size = 10 )
record = rng.choice ( winlosstie, size = 10 )
```

The `size` argument on `rng.choice` allows us to create random matrices too:

```
binary = [ 0, 1 ]
binary_matrix = rng.choice ( binary, size = ( 4, 5 ) )
```

9 Integral estimation

Suppose we need an estimate of a definite integral q of a function $f(x)$, and there is no exact formula for the answer, or we can't think of one. In that case, we could sample the range $[a, b]$ with an array x of n random values to sample the range. If we sum the function values, multiply by $b - a$, and divide by n, we have performed an averaging process that can give a good estimate of q.

```
a = 1.0
b = 10.0
exact = 1.0 / 2.0 * np.log ( ( 2.0 * b + 3.0 ) / ( 2.0 * a + 3.0 ) )
for n in [ 10, 100, 1000 ]:
  x = a + ( b - a ) * rng.random ( n )
  fx = 1.0 / ( 2.0 * x + 3.0 )
  q = ( b - a ) * sum ( fx ) / n
  err = abs ( q - exact )
```

10 Area estimation

We can also use random numbers to estimate the area of a geometric object. As a simple example, consider the region bounded by the ellipse $\frac{x^2}{4} + y^2 = 1$. The exact area of this ellipse is $2 * \pi$. We can surround the ellipse by a rectangle formed by the intervals $[-2, +2] \times [-1, +1]$, whose area is 8. Then we can generate n random points (x, y) in the rectangle. Each such point is actually contained in the ellipse if $\frac{x^2}{4} + y^2 \leq 1$. We count the number of such points as `inside`. Then our estimate for the area of the ellipse is the area of the rectangle times the proportion of points inside the ellipse:

```
exact = 2.0 * np.pi
inside = 0
xlo = -2.0
xhi = 2.0
ylo = -1.0
yhi = 1.0
for n in [ 10, 100, 1000 ]:
  x = xlo + ( xhi - xlo ) * rng.random ( n )
  y = ylo + ( yhi - ylo ) * rng.random ( n )
  fx = 0.25 * x**2 + y**2
  inside = sum ( fx <= 1.0 )
  q = ( xhi - xlo ) * ( yhi - ylo ) * inside / n
  err = abs ( q - exact )
```

This way of estimation is known as **acceptance/rejection**. We start with a set of random points, and then have to apply a test to determine which ones can be accepted (because they are inside the region, in this case). Acceptance/rejection methods have many other applications in which we can generate a random values that approximately have a desired behavior, after which we reject the unacceptable ones.

11 Expected value estimation

A major issue in probability is determining the **expected value** of some process, involving luck, which has a payoff. Playing a slot machine 1000 times, we might ask for our average winnings. Or if we start with $1,000, we might ask how long we could expect to play, on average, if we bet $1 each time. Such questions do not always have a mathematical answer that is easy to determine. Instead, we can try to estimate the expected value by using the computer to "play" the game n times, record each payoff, and then average the result.

As an example, suppose we simply pick two points at random in the unit square, and measure the distance d between them. We know it must be the case that $0 \leq d \leq \sqrt{2}$, and we might guess that the average distance is $\frac{1}{2}$ but it is not easy to work out the true value. On the other hand, it is easy to simulate:

```
exact = ( np.sqrt ( 2.0 ) + 2.0 + 5.0 * np.log ( 1.0 + np.sqrt ( 2.0 ) ) ) / 15.0
for n in [ 10, 100, 1000, 10000 ]:
  x1 = rng.random ( [ n, 2 ] )
  x2 = rng.random ( [ n, 2 ] )
  dx = x1 - x2
  dist = np.sqrt ( dx[:,0]**2 + dx[:,1]**2 )
  dist_ave = np.mean ( dist )
  err = np.abs ( exact - dist_ave )
```

Our numerical estimates correctly suggest that the expected value is not $\frac{1}{2}$!

12 L2 norm $\leq$ Frobenius norm?

The `numpy` library has several options for defining the norm of a matrix A. By default, it uses the Frobenius norm $||A||_F$, which is the square root of the sum of the squares of all the matrix entries. Mathematicians generally respect the $\ell 2$ norm $||A||_2$ instead. Suppose you are told that it is **always** the case that, for any matrix A,

```
||A||_2 \leq ||A||_F
```

This is a useful fact, since $||A||_F$ is easy to compute, and if we simply want to estimate the norm of a matrix-vector product, we could use the result

```
||Ax||_2 \leq ||A||_2 ||x||_2 \leq ||A||_F ||x||_2
```

where the first and last items in this inequality are easy to compute, whereas the middle item might be expensive.

Before doing this, we would like proof of that result. And before expending time working out a mathematical proof, we could easily do some computational checking. If a single computational test fails, then we already know that the statement is wrong, saving us all the effort of trying to prove it, or the embarrassment of jumping ahead and using a (wrong) result without proof. We can easily make a few checks this way. For simplicity, we simply check a lot of 3×3 matrices. We generate our matrices using `rng.standard_normal()` rather than `rng.random()` so that it's likely that our matrices include some negative values, and a wider range.

```
test_num = 1000
true_num = 0
for test in range ( 0, test_num ):
  A = rng.standard_normal ( ( 3, 3 ) )
  A_norm2 = np.linalg.norm ( A, 2 )
  A_normf = np.linalg.norm ( A, 'fro' )
  if ( A_norm2 <= A_normf ):
    true_num = true_num + 1

print ( '  Statement was true ', true_num, ' times out of ', test_num )
```

13 maxcol_minrow versus minrow_maxcol

Given an $m \times n$ matrix A, let u be the m-vector containing the minimum entry in each row, and let v be the n vector containing the maximum entry in each column.

Then define $\alpha = \max(u)$ and $\beta = \min(v)$.

Which of the following is probably true?

$$\alpha = \beta, \text{ always}$$
$$\alpha \leq \beta, \text{ always}$$
$$\alpha \geq \beta, \text{ always}$$
$$\text{Neither } \alpha \text{ nor } \beta \text{ is always dominant.}$$

It is probably not easy to decide this issue at first glance. Follow the previous example, generate 1000 sample 3×3 matrices, and see what your results suggest.