Random numbers
Mathematical Programming with Python
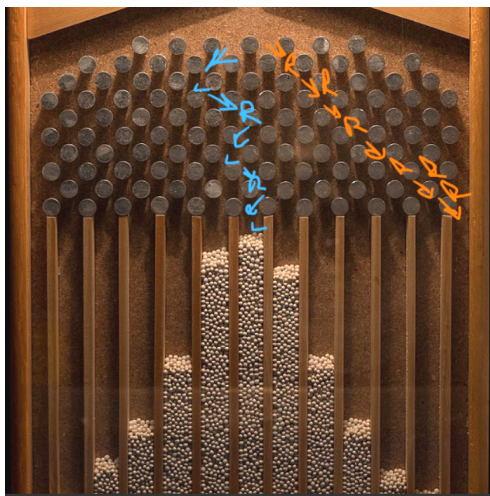
Each ball "chooses" a random path, and yet there is a pattern in the results

Random Numbers

- *The library* `numpy.random` *provides random number generation;*
- *Random numbers sample sets of numbers;*
- *We can sample integers, floats, sets, permutations, polygons.*
- *Sampling usually uniform, but might be weighted or biased.*
- *The random number generator can be restarted at any point;*
- *Many physical and social processes can be modeled using random values.*
- *Refer to the Hill textbook, Chapter 4, Section 5.1, pages 145-148.*

1 Introduction

A random number generator is a function that samples a specified set. Each time the function is called, a new sample is made. On each call, every value in the set may be returned. Each value might have an equal chance of being selected, or the chance may depend on some known weighting function. Theoretically, the results of one call have no effect on the next call.

Actually, the results of random number generators on a computer are completely deterministic; however, the procedure used is deliberately so scrambled that there seems to be no pattern.

Random number generators are used to test algorithms, to simulate physical systems that have a random behavior, to model diseases and social problems in which human behavior is a factor, to estimate areas and volumes of irregular shapes and integrals of complicated functions, or to optimize certain processes. Random numbers are a key component of machine learning algorithms.

2 The Middle Square Method

In the very early days of computing, there was a great need for random numbers that could be created quickly and cheaply as a computer program was running. Because computer memory was tiny, it was not possible to precompute a list of sucn values, they had to be produced one by one, on the fly. John von Neumann came up with an idea known as the "middle square method", for producing a string of random integers of a certain size.

For details, you can refer to an article by Brian Hayes, at
http://bit-player.org/2022/the-middle-of-the-square

The method produced a sequence s of integers with $2d$ digits as follows:

1. Initialize s, an integer with $2d$ digits
2. Replace s by $s * s$, which has no more than $4d$ digits;
3. Drop the last d digits of s;
4. Drop the first d digits of s;
5. s is now the remaining middle $2d$ digits;
6. To get another value, go to step 2.

- We carry out step 3 by integer division by 10^d.
- We carry out step 4 by using modulo 10^{2d}.

As an example, let's work with 4-digit values:

i	s	s^2
0	3647	13300609
1	3006	9036036
2	360	129600
3	1296	1679616
4	6796	46185616
5	1856	3444736
6	4447	19775809
7	7758	60186564
8	1865	3478225
9	4782	22867524
10	8675	75255625

As a homework exercise, you are asked to reproduce this table by implementing the middle square method.

The middle square method was quickly replaced; depending on the initial value, it sometimes started repeating very quickly, or produced results that were not satisfactorily uniform. However, it suggests how a computer might produce a sequence of numbers that at least look random. Notice, in particular, that once we have picked the starting value for s, known as the "seed value", the entire sequence of values is determined.

3 Types of random number needs

Depending on our needs, we will need a random number generator with can randomly select

- a real number r, uniformly, such that $0 < r < 1$;
- a real number r, uniformly, such that $a < r < b$;
- a real number r normally, with mean value μ and standard deviation σ;

- a real number r according to a probability density function;
- a pair (r, s) uniformly inside a rectangle, circle, or other shape;
- an integer, uniformly or with a bias, that is 0 or 1 (flipping a coin);
- an integer i uniformly, such that $i_{min} <= i <= i_{max}$;
- a subset of size k from a set of size n;
- a sequence of size k with values selected from a set of size n (repetitions allowed);
- a permutation of n values;

To say an object is selected "uniformly" from a set is to say that every object has an equal chance of being selected. Selecting an object "normally" gives a strong preference to the mean value, and to objects within one standard deviation of it. To select an object according to a probability density function means there is a rule that specifies exactly how likely every possible value is.

Python makes available a set of random number generators that can handle all these cases.

4   The numpy random functions

The `numpy` library includes a `.random` sublibrary containing the functions that can handle our random number needs. Using the statement

```
import numpy as np
```

we would refer to `np.random.random()`; if we use

```
from numpy.random import random
```

then we can use the shortened name `random()`.

The set of random number functions includes:

- `seed(value)`: initializes the random number generator.
- `random(n)`: n uniform random values in [0,1];
- `uniform(a,b,n)`: n uniform random values in [a,b];
- `randint(low,high,n)`: selects an integer from a range;
- `random.choice(list,n)`: selects a subset or sequence from a set;
- `randn(n)`: a normal random value;

For instance, here's a small script that sets the seed to 42, then defines a random number, a random vector, and a random matrix:

```
from numpy.random import seed, rand

seed ( 42 )
x = random ( )         # a number (no brackets, no index)
y = random ( 1 )       # an np array, a "vector" of 1 value
v = random ( 5 )       # an np array, a vector of 5 values
A = random ( 3, 2 )    # an np array, a 3x2 matrix
```

Setting the seed allows you to control how the random number sequence begins. Every time you run this script, you will restart the random number generator at the same position, and hence get the same "random" values. This is good for debugging. If you omit the call to `seed()`, then each time you run the script, the sequence of values will be different.

5   Simple tests

One thing we expect from a random number generator is that the values returned are uniform. Every value should be equally likely to be chosen. So we could expect that if we ask for a thousand or ten thousand such values, they would be spread out over the unit interval a bit like snowfall. If we simply plot 1000 values as dots, we won't really see a useful pattern. But a histogram, which divides the interval up into bins, can help us to see how the values are arranged.

```
import matplotlib.pyplot as plt
import numpy as np

n = 1000
plt.clf ( )
r = np.random.random ( n )
plt.hist ( r, bins = 50 )
plt.grid ( True )
plt.savefig ( 'random_histogram.png' )
plt.show ( )
plt.close ( )
```
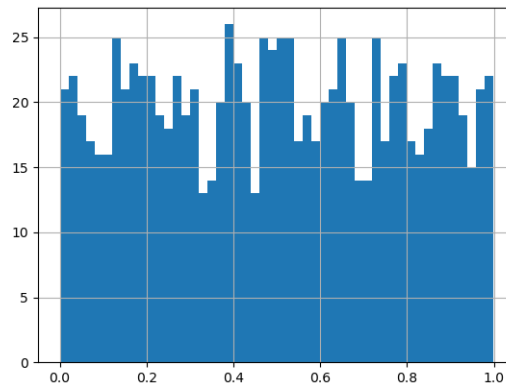

The results look very ragged. However, it's worth while to try a larger value of n to see whether we get smoother results.

Another test is to compare the average value of our sample to the average of the set of real numbers in the unit interval:

$$\mu = \int_0^1 x dx = \frac{1}{2}$$

A third test compares the variance of our sample to the variance of the set of real numbers in the unit interval:

$$var = \int_0^1 (x - \frac{1}{2})^2 dx = \frac{1}{12}$$

We can compute these quanties with the `numpy` functions `mean()` and `var()`:

```
import numpy as np
n = 1000
x = np.random.random ( n )
var_rand = var ( x )
mean_rand = mean ( x )
```

Here, my results for 1000 samples were:

```
mean(r) =  0.4949916524887399
var(r)  =  0.08519763801660969
```

Following these results, we have some belief that our random number generator is well behaved.

6 Extending random() to other problems

Often, we want our random numbers to be in some other interval, $[a, b]$. This can be done by calling `random()` to get values in [0,1], multiplying by $(b - a)$ and adding a.

```
a = 10
b = 15
x = a + ( b − a ) * random ( )    # Returns a number
y = a + ( b − a ) * random ( 5 ) # Returns an np.array of 5 values
```

but there is also a `numpy` function to do this automatically:

```
a = 10
b = 15
x = np.random.uniform ( a, b )
y = np.random.uniform ( a, b, 5 )
```

If we need random points (x, y) inside a rectangle $[a, b] \times [c, d]$, we can simply ask `np.random.uniform()` for $a <= x <= b$ and then $c <= y <= d$.

There are geometric tricks that allow us to uniformly sample triangles and general polygons, circles and ellipses, but the details of these methods will have to wait for now.

The `rand()` function can also be used for coin-flipping problems. We may need to flip a coin a thousand times. If the coin is fair, then we can simply generate 1000 random values, and count the values below or above 0.5 as tails and heads:

```
n = 1000
r = random ( n)
t = sum ( r < 0.5 )
h = sum ( 0.5 <= r )
print ( '  Relative frequency of tails = ', t / n )
print ( '  Relative frequency of heads = ', h / n )
```

With this idea, we can easily handle a coin that has a bias. Say we have weighted a coin to come up tails 40% and heads 60% of the time. We simply change the code as follows:

```
t = sum ( r < 0.4 )
h = sum ( 0.6 <= r )
```

Similarly if we had a situation with three outcomes, whose individual probabilities were p_0, p_1, p_2 (which should sum to 1!) then we would count the number of times each outcome occurred by something like this:

```
n0 = sum ( r < p[0]                          )
n1 = sum (      p[0] <= r & r < p[0] + p[1]      )
n2 = sum (                      p[0] + p[1] <= r )
```

Remember, we have to use & when combining logical results involving `numpy` arrays. A similar idea can be used when there are many possible outcomes; however, usually in such cases the outcomes are equally likely, and a random integer based approach is easier to work with.

7 Uniform random integers

Suppose we are trying to randomly pick a day of the year, a card in a deck of 52, or a five digit ZIP code. Assuming that every element in the group is to be selected with equal likelihood, then we can carry out our task by asking for a random integer k between limits: $k_min \le k < k_max$.

The appropriate function to use is `np.random.randint()`, which allows us to specify the bounds and n, the number of samples we want (defaulting to 1):

```
k = np.random.randint ( low = kmin, high = kmax, size = n )
```

Any of the calls

```
k = randint ( 10, 20 )                    # keywords optional
k = randint ( low = 10, high = 20 )       # keywords used
k = randint ( low = 10, high = 20, size = 1 )  # size specified
```

returns 1 random integer k such that $10 \le k < 20$. To get an array of, say, 15 such values, try either:

```
k = randint ( 10, 20, 15 )
k = randint ( low = 10, high = 20, size = 15 )
```

Similarly, you could request a binary string (that is, 0's and 1's) of length 50:

```
b = randint ( 0, 2, 50 )       # Remember, 0 <= b[] < 2
```

Note that when `randint()` returns an array of n values, it is quite possible that some values will be repeated. This is because we are performing what is technically called "selection with replacement"; even if a value is picked for an entry of our list, it can be picked again for a later entry of the same list. This happens, obviously, in the case of the binary string, where you only have two choices for 50 slots; but it also can happen when we are picking 5 values from an interval of 100.

This means that `randint()` would **not** be a good choice if you are trying to model a card game, in which the 52 items can each be chosen only once or not at all.

8 Selection without replacement

When modeling a card game, in which a small selection of cards, maybe 5, is made from a deck of 52, you want to guarantee that your random list does not have any repeated values. You are asking for a random *subset* of the cards. In Python, we can select a few items from a set at random by using the `np.random.choice()` function.

This function has the form

```
subset = np.random.choice ( my_set, n, replace = False )
```

where `my_set` is an array of values, `n` is the desired number of selections, and `subset` contains the chosen values. For this function, the value of `n` can't be larger than the number of elements in the set. Here are two examples, for small and large sets:

```
small_set = range(0,52)                              # small_set = [ 0, 1, ..., 51 ]
subset = np.random.choice ( small_set, 5, replace = False )  # Pick 5 from the list
big_set = range(0,1000)                              # big set = [ 0, 1, ..., 999 ]
subset = np.tandom.choice ( big_set, 5, replace = False )    # Pick 5 items
```

The default value of `replace` is `True`. Setting `replace = True` or using the default value allows items to be selected more than once.

9 Random permutations

We might model a deck of cards as an array that runs from 0 to 51. In real life, we shuffle the deck and then deal out cards from the top one at a time. To simulate this process in Python, we can use the function

```
deck = np.arange ( 0, 52 )
new_deck = np.random.shuffle(deck)
```

If we only wanted 5 cards at random, we could have called `np.random.choice()`. Here, we are shuffling the entire deck; the top five cards would then be available as our random hand.

10 Normal random values

```
x = np.random.normal ( loc = mu, scale = sd, size = n )
```

returns `n` samples of the normal distribution with given mean `mu` and standard deviation `sd`.

The normal distribution describes a process which returns values tending to a given average but having errors or deviations of typical size `sd`.

When `mu=0` and `sd=1`, the underlying process is described as the standard normal distribution. When a normal distribution function is plotted, it tends to look like a "bell curve".

As we did for the uniform random number function, we can also compute 1000 normal random values, and determine the mean and standard deviation.

```
import matplotlib.pyplot as plt
import numpy as np

n = 1000
plt.clf ( )
r = np.random.normal ( loc = 2.0, scale = 1.5, size = n )
plt.hist ( r, bins = 50 )
plt.grid ( True )
plt.savefig ( 'normal_histogram.png' )
plt.show ( )
plt.close ( )

normal_mean = np.mean ( r )
normal_stdev = np.std ( r )
print ( '  normal mean  = ', normal_mean )
print ( '  normal stdev = ', normal_stdev )
```
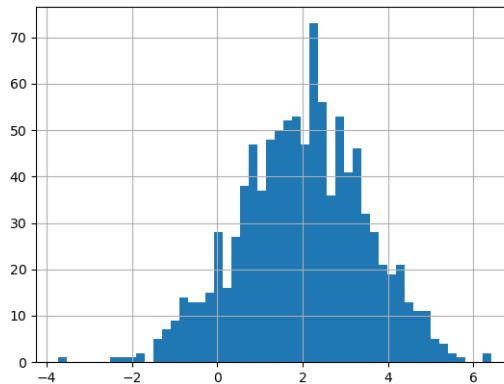
11 Random Integers Solve Newton's Dice Problem

According to Wikipedia: *In 1693, Samuel Pepys and Isaac Newton corresponded over a problem posed to Pepys by a school teacher named John Smith. The problem was:*

Which of the following three propositions has the greatest chance of success?

1. *Six fair dice are tossed independently and at least one "6" appears.*
2. *Twelve fair dice are tossed independently and at least two "6"s appear.*
3. *Eighteen fair dice are tossed independently and at least three "6"s appear.*

Pepys thought that the third outcome had the highest probability, but Newton thought it was the first outcome

While the correct answer can be determined using probability, we can quickly come up with a computational approach that helps us to guess what the answer will be. We can repeat each experiment 1000 times, and report the frequency with which the desired number of 6's occur. Here's how we would do it for the first case:

```
freq = 0
test_num = 1000
for test in range ( 0, test_num ):
    dice = np.random.randint ( low = 1, high = 7, size = 6 )
    if ( 1 <= np.sum ( dice == 6 ) ):
        freq = freq + 1
print ( ' Case 1 probability estimate is ', freq / test_num )
```

The exercises ask you to estimate the probabilities for all three cases, and decide who was right, Newton. Pepys, or neither of them!

12 Estimating the probability of a straight in poker

A standard poker deck consists of 52 cards. The cards are organized into four suits: hearts, clubs, diamonds, and spades. Each suit contains 13 cards, having the ranks 1 through 13. After the deck is randomly shuffled, a player received 5 cards from the deck. Certain arrangements of cards are more valuable than others, and the players proceed to bet on who has the best hand.

One fairly rare arrangement is known as a *straight*. It is any hand of 5 cards which can be arranged to form a sequence, such as 4, 5, 6, 7, 8. In this case, only the ranks of the cards matter, while the suits may be different.

We wish to estimate the probability of getting a straight by simulating a random poker hand. It is natural to start this process by defining `deck` as a list of the values 0 thought 51, and then using `np.random.choice()` to select from `deck` 5 different values:

```python
deck = range ( 0, 52 )      # deck = [ 0, 1, 2, 3, ..., 51 ]
hand = np.random.choice ( deck, 5, replace = False )
```

If our hand is the values {7, 8, 9, 10, 11} then we can see right away that we have a straight. However, there are 4 cards of rank 7, namely 7, 7+13=20, 7+26=33, and 7+39=46, from the suits of hearts, clubs, diamonds and spades. We need to be able to turn the card index into a rank. That's easy, because as you can guess, it's just the remainder after division by 13.

```python
hand = ( hand % 13 )
```

You should convince yourself that this adjustment correctly ranks all 52 cards.

If this was a real poker game, the next thing you might do is sort your cards by rank. We can do the same thing in Python with the `.sort()` method:

```python
hand.sort()
```

And now we can easily check whether we have a straight:

```python
def is_straight ( hand ):
  value = True
  for i in range ( 0, 4 ):        # Note the range is NOT ( 0, 5 )!
    if ( hand[i+1] != hand[i] + 1 ):
      value = False
      break
  return value
```

So now we have the tools we need to estimate the probability of a straight. We simply deal out a large number of hands, count the number of occurrences of a straight, and take the ratio. The exact value is $10,240/2,598,960 \approx 0.0039$ which means that about 4 times in 1000 you would be dealt a straight.

The exercises ask you to verify this claim.