2. Random Number Generator

Overview

In this lecture, we discuss the basics of random number generation algorithm. By the end of the lecture, you should be able to:

(1) understand the basics of how a random number can be generated

(2) able to implement a simple random number generator

(3) understand np.random.rand()

Random number generator

A **random number generator** (**RNG**) is a device that generates a sequence of numbers or symbols that cannot be reasonably predicted better than by a random chance.

There are two ways to generate a random number:

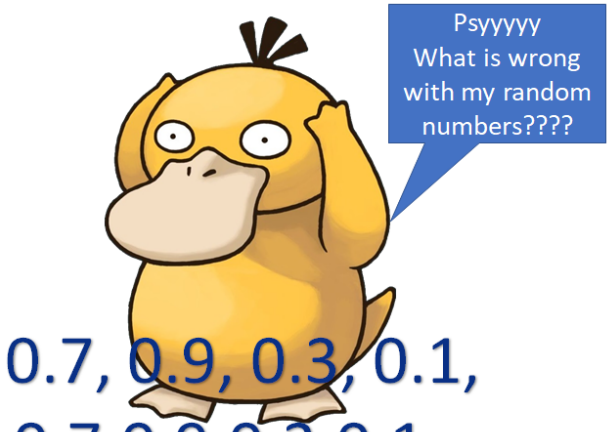
(1) Hardware (True) random number generator

The first way of generating random numbers relies on physical phenomena. For example, the next number generated can depend on the level of background noise in your office, the decay of radioactive sources, or even the movement of your mouse. With well-control designs, these sources will be unpredictable and not repetitive. As a result, the numbers will not start repeating the patterns after a cycle.

True random number generator is widely used in settings that rely on high-level security, for example, banks, gambling, cybersecurity, or national defense.

(2) Pseudo-random number generator

The second way to generate random numbers is based on computer algorithms. Using these algorithms, we are able to generate numbers that seem to be random, although they will be repeating themselves after a cycle. A good algorithm should be able to produce a long sequence of numbers that follows a random order without repeating themselves for a long sequence.



0.7, 0.9, 0.3, 0.1,
0.7,0.9,0.3,0.1…

A simple RNG algorithm

One algorithm to generate a random number is called "Linear congruential generator"(LCG).

A simple computer-based algorithm to generate random numbers is as follows:

- Step 0: initialize $x = x_0$. x_0 is our seed.
- Step 1:
  - We update $x = (a \times x + c)\%m$.
  - We generate the first random number using x/m
- Step 2:
  - We update $x = (a \times x + c)\%m$.
  - We generate the second random number using x/m
- ...
- Step n:
  - We update $x = (a \times x + c)\%m$.
  - We generate the nth random number using x/m

Where

- m is the "modulus" and set to be a positive number
- a is the "multiplier" and should satisfy $0 < a < m$
- c is the "increment" and should satisfy $0 \leq c < m$
- x_0 is the seed and should satisfy $0 \leq x_0 < m$

Although this algorithm is very simple, with the appropriate choice of parameters, the period is known and long. For example, For C++11, c, a, and m are set equal to 0, 7^5 and $2^{31} - 1$ respectively.

The random numbers are the foundations for the simulation. We rely on random numbers to generate samples from different distributions. In the following lectures, we will cover different techniques to convert this random number to other numbers that follow our desired distributions.

`for` loop

Writing a "for" Loop

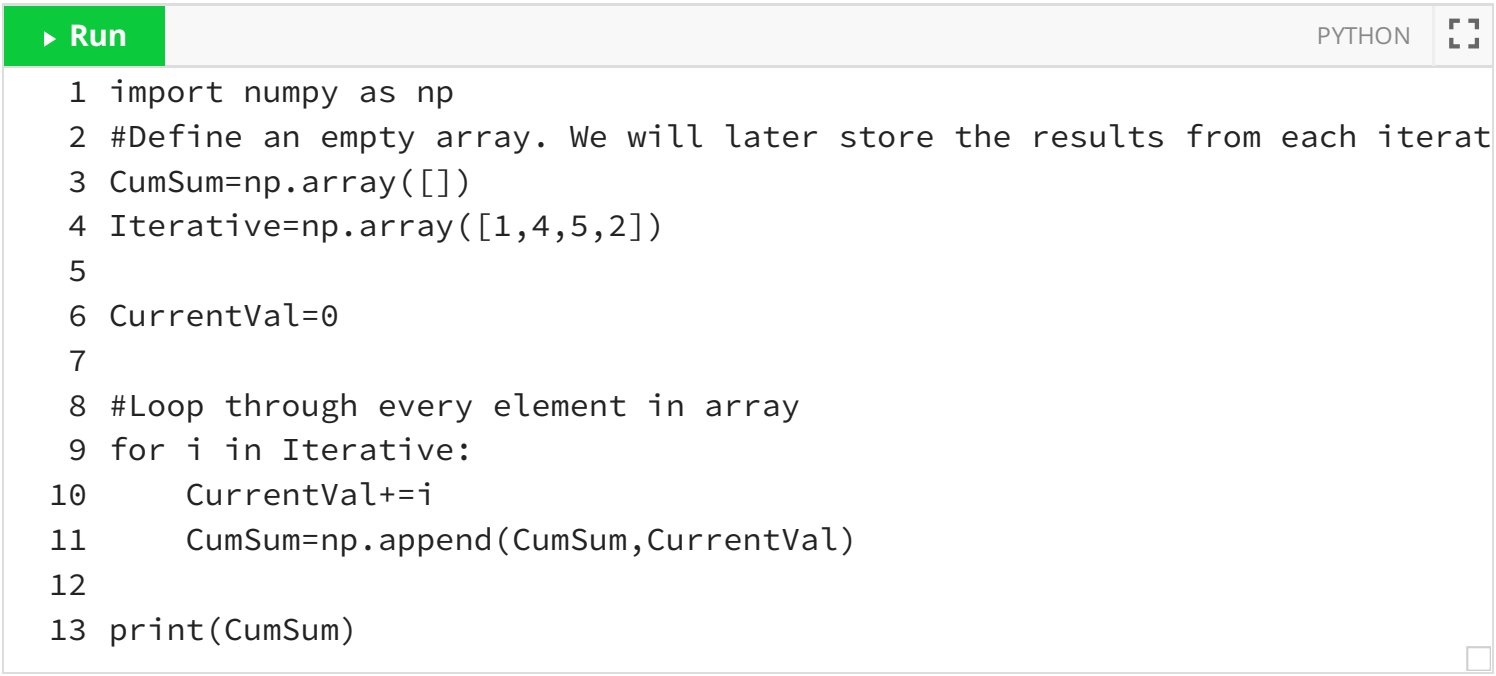
We can use loops when we want to do the same task multiple times or a sequence of tasks iteratively. The structure of a "for" loop is as follows:

```
for i in array/list:
  operations
```

Loops will do the following:

1. Assign i=first element of the array
2. Run the operations
3. Assign i=second element of the array
4. Run the operations
5. Assign i=third element of the array
6. Run the operations
7. Repeat, until i loops through all the values in the array.

In the following example, we perform a cumulative summation for sequence 1, 4, 5, 2.



Notice that, we used `np.append(a1, a2)` function in the last step. The purpose of this statement is to append a2 (a value, an array, or a list) to the back of another array.

Run PYTHON

```python
import numpy as np
a1=np.arange(3)
a2=np.arange(2,5)
np.append(a1,a2)
print(a1)
#save the updated array
a1=np.append(a1,a2)
print(a1)
```

RNG visual

The Jupyter Notebook on the right-hand side visualizes the random numbers collected using the simple algorithm.

The coding for this Jupyter Notebook is not required. Please just focus on the plots.

Built-in RNG

For this course, we will rely on np.random.rand(num) to help us generate random samples.

This generator is implemented based on Mersenne Twister and implemented using C. The algorithm is more complicated than Linear Congruential Generator. It is the most widely applied Psuedo-RNG and has a long period equals $2^{19937} - 1$. The algorithm itself is out of the scope for this course. For those who are interested in the algorithm, please check here.

▸ Run PYTHON

```
1 print(2**19937-1)
```

Examples :

▸ Run PYTHON

```
1 import numpy as np
2 print(np.random.rand())
3 print(np.random.rand(10))
```

`np.random.rand()` will use a default seed.

We can also, instead define a seed, so that we are able to reproduce the result in every iteration. The stream of random numbers will be fixed after we fix the seed.

▸ Run PYTHON

```
1 import numpy as np
2 np.random.seed(10)
3 print(np.random.rand(10))
4 print(np.random.rand())
5 np.random.seed(10)
6 print(np.random.rand(11))
7
8
```

To provide more flexibility, we can fix the state of the RNG using `RNG=np.random.RandomState(seed)` , then use `RNG.rand()` to generate a stream of random numbers.

Run PYTHON

```python
import numpy as np
RNG1=np.random.RandomState(seed=10)
RNG2=np.random.RandomState(seed=10)
print(RNG1.rand(5))
print(RNG2.rand(5))
print(RNG1.rand(5))
```

Evaluating samples

`Numpy` provides several functions that come handy when studying sample values

Where, `a` is an array that contains all the sample values. `q` is a number between 0 and 100 inclusive, indicating the percentile. `ddof` stands for Delta Degrees of Freedom. To compute sample mean and sample standard deviation, set `ddof=1`

Function	**Description**
np.mean(a)	Sample mean
np.std(a,ddof=1)	Sample standard deviation
np.var(a,ddof=1)	Sample variance
np.percentile(a,q)	Sample percentile
np.min(a)	Sample minimum
np.max(a)	Sample maximum