% ISOM 3025 Lect3 % Yi Ding % January 2023

Lecture 3: Random number generating with Python

Outlines

In this lecture:

- Go over the jupyter notebook enviroment in Python programming
- Random number generating with Python

Setting up the programming enviroment

We use python and jupyter notebook for case study. Also, you will do quiz using jupyter notebook in class, so I highly recommend you setting up the enviroment in advance if you haven't do so.

Jupyter is a software development environment where you can run Python code, including the examples in our course, and write your own code.

1. Installing python and jupyter notebook

Download Anaconda

I suggest you to use Anaconda, which is a free Python distribution that includes all the libraries you need for this course. Anaconda is available for Linux, macOS, and Windows. By default, it puts all files in your home directory, so you don’t need administrator (root) permission to install it, and if you have a version of Python already, Anaconda will not remove or modify it.

Start at https://www.anaconda.com/download. Download the installer for your system and run it. I recommend you run the installer as a normal user, not as administrator or root.

I suggest you accept the recommended options. On Windows you have the option to install Visual Studio Code, which is an interactive environment for writing programs. You won’t need it for this book, but you might want it for other projects.

By default, Anaconda installs most of the packages you need, but there are a few more you have to add. Once the installation is complete, open a command window. On macOS or Linux, you can use Terminal. On Windows, open the Anaconda Prompt that should be in your Start menu.

Run the following command (copy and paste it if you can, to avoid typos):

```
conda install jupyter pandas sympy
conda install beautifulsoup4 lxml html5lib
conda install pint
```

To install any packages xxx in the future:

```
conda install xxx
```

2. Running Jupyter

If you have not used Jupyter before, you can read about it at https://jupyter.org.

To start Jupyter on macOS or Linux, open a Terminal; on Windows, open Git Bash. Use cd to "change directory" into the directory that contains the notebooks. Then launch the Jupyter notebook server:

Example

```
cd ISOM3025
jupyter notebook
```

Note: If you can not open jupyter notebook in cmd, try open anaconda promt and open it there. Alternative, try solution here: https://stackoverflow.com/questions/44515769/conda-is-not-recognized-as-internal-or-external-command

Jupyter should open a window in a browser, and you should see the list of notebooks in your repository. Click on the first notebook, and follow the instructions to run the first few "cells". The first time you run a notebook, it might take several seconds to start, while some Python files get initialized. After that, it should run faster.

You can also launch Jupyter from the Start menu on Windows, the Dock on macOS, or the Anaconda Navigator on any system. If you do that, Jupyter might start in your home directory or somewhere else in your file system, so you might have to navigate to find the directory with the notebooks.

---

3. Opening your working .ipynb file

To start with, you can open any working file in your home directory. Or you can creat working file by click the drawdown button "new" on topright corner, and then click Python 3 (ipykernel).

Generating random variables

Example:

```
1. Specifying a "seed", say 1234
2. generating random integer between 1-10
```

In [25]:
```
import random
random.seed(1234)
random.randint(1,10)
```
Out[25]:
```
8
```
In [ ]:
```

```
What will happen if we generate a random integer again

In [28]:
```
random.randint(1,  10)
```
Out[28]:
```
2
```
What will hapen if we restate the seed

In [30]:
```
random.seed(1234)
random.randint(1,  10)
```
Out[30]:
```
8
```
The random module is based on the Mersenne Twister algorithm, which was originally developed to produce inputs for Monte Carlo simulations. The Mersenne Twister algorithm is a pseudo-random number generator (PRNG) that produces almost uniform numbers suitable for a wide range of applications.

random.randint function generates random integers. The arguments for randint() are the values of the range, including the extremes. The numbers may be negative or positive, but the first value should be less than the second.

The seed function initializes the basic random number generator. Then the randomint produces same set of random data.

The pseudo-random number generator (PRNG)

The generation of *real* random sequences using deterministic algorithms is impossible: at most, *pseudo-random* sequences can be generated. These are, apparently, random sequences that are actually perfectly predictable and can be repeated after a certain number of extractions.

A **PRNG** is an algorithm designed to output a sequence of values that appear to be generated randomly.

The pros and cons of a random number generator

A random number generation routine must be the following:

- Replicable
- Fast
- Not have large gaps between two generated numbers
- Have a sufficiently long running period
- Generate numbers with statistical properties that are as close as possible to ideal ones

The most common cons of random number generators are as follows:

- Numbers not uniformly distributed
- Discretization of the generated numbers
- Incorrect mean or variance
- Presence of cyclical variations

Linear congruential generator

One of the most common methods for generating random numbers is the Linear Congruence Generator (LCG). The theory on which it rests is simple to understand and implement. It also has the advantage of being computationally light. The recursive relationship underlying this technique is provided by the following equation:

$$x_{k+1} = (ax_k + c) \bmod m,$$

where

- $x_1, \ldots,$ is a series of generated numbers,
- a is the multiplier (non-negative integer),
- c is the increment (non-negative integer),
- m is the mode (non-negative integer),
- x_0 is an initial value (seed or non-negative integer).

The modulo function, denoted by *mod*, results in the remainder of the Euclidean division of the first number by the second. For example, 18 mod 4 gives 2 as it is the remainder of the Euclidean division between the two numbers.

The linear congruence technique has the following characteristics:

- It is cyclical with a period that is approximately equal to m.
- The generated numbers are discretized.

Now we are ready to generate some random numbers by ourselves!

In [31]:
```
import numpy as np
a = 2
c = 4
m = 10
x = 3
for i in range (1,20):
    x= np.mod((a*x+c),m)
    print(x)
```

```
0
4
2
8
0
4
2
8
0
4
2
8
0
4
2
8
0
4
2
```

numpy is a library of additional scientific functions of the Python language, designed to perform operations on vectors and dimensional matrices. numpy allows you to work with vectors and matrices more efficiently and faster than you can do with lists and lists of lists (matrices). In addition, it contains an extensive library of high-level mathematical functions that can operate on these arrays.

np.mod function returns the remainder of a division when given a dividend and divisor.

Random numbers with uniform distribution

A sequence of numbers uniformly distributed between [0, 1] can be obtained using the following formula:

$$U_n = \frac{x_n}{m}.$$

Important note To make this work, you need to choose a large value of m, you can reduce both the phenomenon of periodicity and the problem of generating rational numbers. Generally, a value of m is m ≥ 109 so that the generated numbers constitute a dense subset of the interval, [0, 1].

Learmonth-Lewis generator. This is a generator in which the parameters assume the following values:

- $a = 75$
- $c = 0$
- $m = 2^{31} - 1$

In [23]:

```
import numpy as np
a = 75
c = 0
m = 2**(31) -1
x = 0.1

for i in range(1,10):
    x= np.mod((a*x+c),m)
    u = x/m
    print(u)
```

```
3.4924596564343477e-09
2.619344742325761e-07
1.9645085567443206e-05
0.0014733814175582405
0.11050360631686804
0.28777047376510245
0.5827855323826827
0.708914928701201
0.1686196525900716
```

Exploring generic methods for a generic distribution

We are now able to generate uniform distributions in the range of [0, 1]. Next, we will examine some methods that allow us to derive a generic distribution starting from a uniform distribution of random numbers.

The inverse transform sampling method

By having a PRNG with continuous and uniform distributions in the range of [0, 1], it is possible to generate continuous sequences with any probability distribution using the inverse transform sampling technique.

Consider a continuous random variable, x, having a probability density function of $f(x)$. The corresponding distribution function, $F(x)$, is determined for this function, as follows:

$$F(x) = \int^{x} f(x)dx.$$

The inverse function is then determined, $x = F^{-1}$.

Importantly, **the CDF follows uniform distribution (please check and prove it yourself!) on [0,1]**, hence we can first generate uniform distribution and then use the inverse function F^{-1} to obtain the corresponding x.

Example: Exponential distribution

$$f(y) = \lambda e^{-\lambda y}.$$

The CDF is

$$F(y) = 1 - e^{-\lambda y}.$$

The inverse function of the CDF is

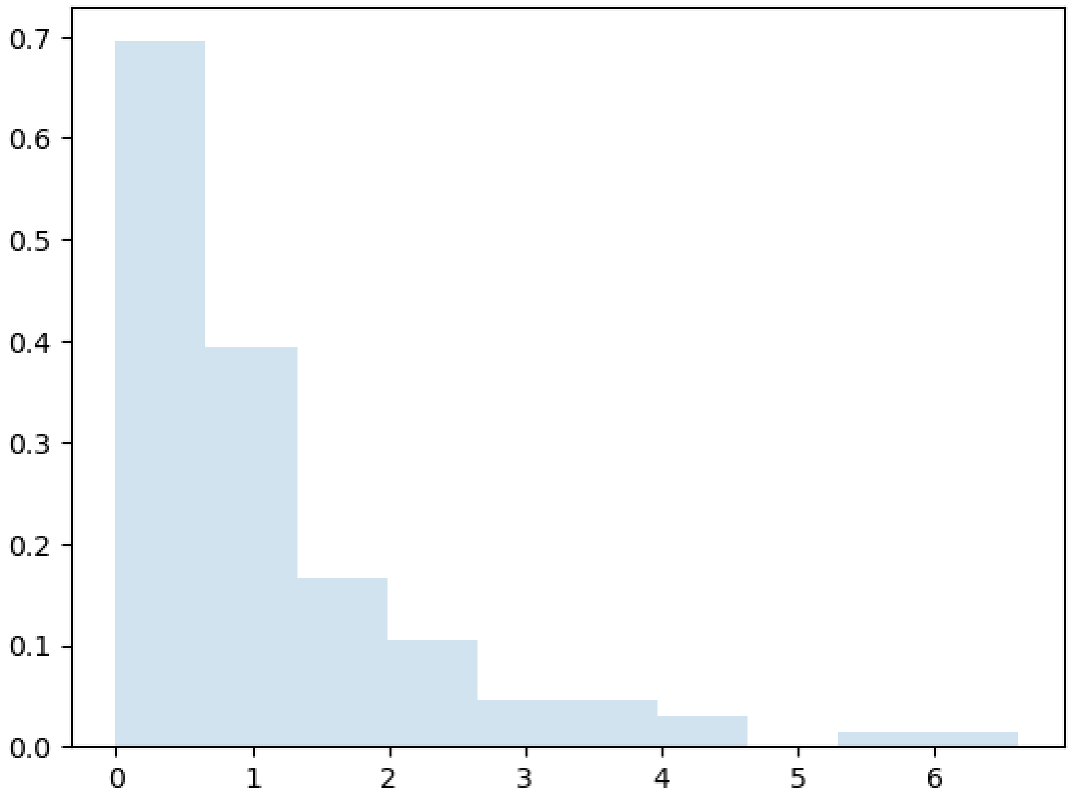
$$y = -\frac{1}{\lambda}\log(1 - F).$$

What is the CDF of uniform distribution between 0 and 1?

Proof: the CDF of any random variable follows uniform [0,1]. For simplicity, suppose that the CDF is strictly increasing. %

In [39]:
```
import numpy as np
import matplotlib.pyplot as plt
a = 75
c = 0
m = 2**(31) -1
x = 0.1
lam=1
ylist=[]
for i in range(1,101):
    x= np.mod((a*x+c),m)
    F = x/m
    y = -1/lam*np.log(1-F)
    ylist.append(y)
#     print(y)

plt.figure()
plt.hist(ylist,density=True,histtype='stepfilled',alpha=0.2)
plt.show()
```



- The matplotlib.hist() function draws a histogram. In a histogram, each rectangle has a non-random length equal to the width of the class it represents. The height of each rectangle is equal to the ratio between the absolute frequency associated with the class and the amplitude of the class, and it can be defined as frequency density.
- density=True: the function return the counts normalized to form a probability density.
- histtype='stepfilled': This parameter defines the type of histogram to draw. The stepfilled value generates a line plot that is filled by default.
- alpha=0.2: This is a float value that defines the characteristics of the content (0.0 is transparent and 1.0 is opaque).

See more examples and ready to use of random variable generation codes in https://realpython.com/python-random/

In summary, in this lecture:

```
We learn algorithms of generating random integers within a range,
uniform distribution and a general distribution using Python.
```

References: GC Chapter2

Some useful on-line materials:

- on-line course material for simulation with Python: https://allendowney.github.io/ModSimPy/index.html
- jupyter notebook tutorial: https://www.dataquest.io/blog/jupyter-notebook-tutorial/

NEXT

```
Monte Carlo Simulation
Example 1
```

```
In [ ]:
```