**Hands on Markov Chains example, using Python**

**Demystifying Markov Chain one line of code at a time.**

[Piero Paialunga](https://piero-paialunga.medium.com/?source=post_page-----8138bf2bd971--------------------------------) Dec 31, 2021

https://towardsdatascience.com/hands-on-markov-chains-example-using-python-8138bf2bd971

When I started studying Physics, I didn’t like the concept of **probability**. I was so pumped by the idea that with Physics you can model the entire world that the idea of uncertainty made me furious :)

The truth is that when we want to study real phenomena we must, sooner or later, deal with a certain level of uncertainty. And the only way to deal with it is by gaining an accurate estimate of the probability that rules our process.

**Markov Chains are an excellent way to do it**. The idea that is behind the Markov Chains is extremely simple:

Everything that will happen in the future only depends on what is happening right now

In mathematical terms, we say that there is a sequence of stochastic variables X\_0, X\_1, …, X\_n that can take values in a certain set A. Then we say that, if the sequence of an event is a Markov Chain, we have:



Image generated by me using [LaTeX](https://latex2png.com/)

And it may sound complicated but it is nothing more than the concept expressed above.

Another assumption that is made is that the equation is valid for every step (not only the last one), and that the probability is always the same (even though, formally, this is true only for **homogeneous Markov chains**).

Now, the set of possible states A is usually indicated as the sample space S, and you can describe the probability of going from a state x in S to a state y in S with the so called **transition probability.**

But I promised you that this would have been an “Hands on” article, so let’s start visualizing these concepts!

To be fair, Python is not the best environment to perform numerically simulations. Professional researchers use much more complex and in some way reliable languages like **C** or **Fortran.**

Nonetheless, the goal of this blog is to introduce some very simple concepts and using Python can make this learning process easier.

So let’s dive in: this is what you’ll need:

|  |
| --- |
| import matplotlib.pyplot as plt  import numpy as np  import pandas as pd  import seaborn as sns  plt.style.use('ggplot')  plt.rcParams['font.family'] = 'sans-serif'  plt.rcParams['font.serif'] = 'Ubuntu'  plt.rcParams['font.monospace'] = 'Ubuntu Mono'  plt.rcParams['font.size'] = 14  plt.rcParams['axes.labelsize'] = 12  plt.rcParams['axes.labelweight'] = 'bold'  plt.rcParams['axes.titlesize'] = 12  plt.rcParams['xtick.labelsize'] = 12  plt.rcParams['ytick.labelsize'] = 12  plt.rcParams['legend.fontsize'] = 12  plt.rcParams['figure.titlesize'] = 12  plt.rcParams['image.cmap'] = 'jet'  plt.rcParams['image.interpolation'] = 'none'  plt.rcParams['figure.figsize'] = (12, 10)  plt.rcParams['axes.grid']=False  plt.rcParams['lines.linewidth'] = 2  plt.rcParams['lines.markersize'] = 8  colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',  'xkcd:scarlet'] |

So it is a bunch of mainstream libraries like **pandas**, **matplotlib, seaborn and numpy.**

Let’s start from the simplest scenario ever:

**1. Random Walks**

The **simple** random walk is an extremely simple example of a random walk.

The first state is 0, then you jump from 0 to 1 with probability 0.5 and jump from 0 to -1 with probability 0.5.

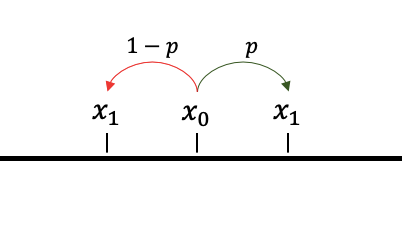


Image made by me using Power Point

Then you do the same thing with x\_1, x\_2, …, x\_n.

You consider S\_n to be the state at time n.

It is possible to prove (and it is actually very easy) that the probability of being in a certain state, i.e. an integer number x, at time t+1 only depends on the state at time t. In short words, **it is a Markov Chain.**

So this is how to generate it:

|  |
| --- |
| start = 0  x = []  n = 10000  for i in range(n):  step = np.random.choice([-1,1],p=[0.5,0.5])  start = start + step  x.append(start) |

And it is the result:

|  |
| --- |
| plt.plot(x)  plt.xlabel('Steps',fontsize=20)  plt.ylabel(r'$S\_{n}$',fontsize=20) |

Now the idea of the random walk is to simulate what is going to happen if we decide to start from a point and randomly choose to go up or down by flipping a perfect coin.

This process is pretty simple, yet so much interesting in terms of its theoretical applications and properties.

The first reasonable extension of this process is to consider a random walk but with a non-perfect coin. It means that the probability of going up is not the same probability of going down. This is called **biased random walk.**

Let’s consider the following couple of probabilities:

|  |
| --- |
| [0.1,0.9] , [0.2,0.8], [0.4,0.6], [0.6,0.4], [0.8,0.2],[0.9,0.1] |

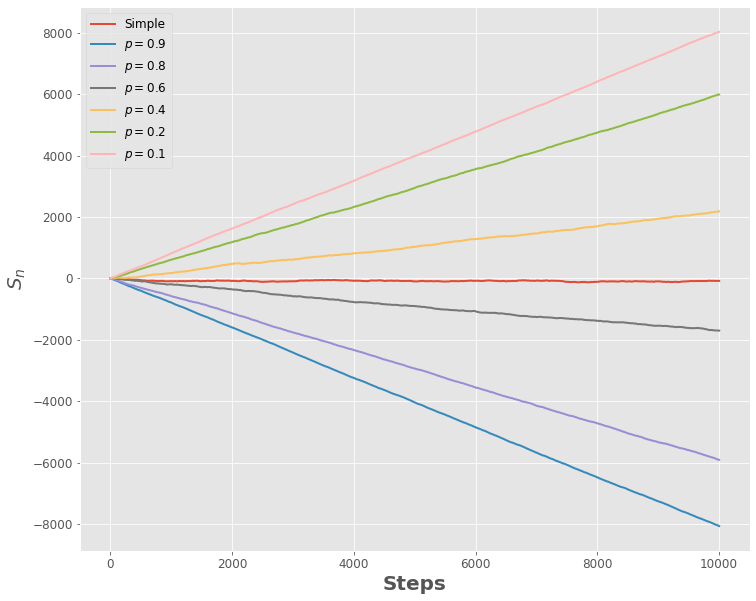
So we have 6 possible random walks. Note that the probability has to be 1, thus it is sufficient to consider the “up” or “down” probability.

Here is how you do it:

|  |
| --- |
| x = []  p = [[0.5,0.5],[0.9,0.1],[0.8,0.2],[0.6,0.4],[0.4,0.6],[0.2,0.8],[0.1,0.9]]  label\_p = ['Simple',r'$p=0.9$',r'$p=0.8$',r'$p=0.6$',r'$p=0.4$',r'$p=0.2$',r'$p=0.1$']  n = 10000  x = []  for couple in p:  x\_p = []  start = 0  for i in range(n):  step = np.random.choice([-1,1],p=couple)  start = start + step  x\_p.append(start)  x.append(x\_p) |

And this is what happens when we visualize it.

|  |
| --- |
| i=0  for time\_series in x:  plt.plot(time\_series, label = label\_p[i])  i=i+1  plt.xlabel('Steps',fontsize=20)  plt.ylabel(r'$S\_{n}$',fontsize=20)  plt.legend() |



**2. Gambler’s Ruin Chain**

Another simple way to extend the random walk is the gambler’s ruin chain.   
Conceptually, it is very similar to the random walk: you start from a state x and you can go to a state y=x+1 with probability p or to a state y=x-1 with probability 1-p.

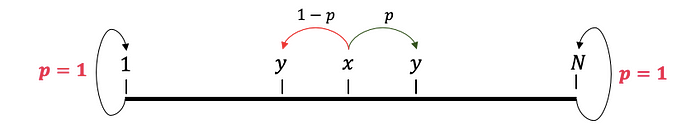


Image made by me using Power Point

The interesting part is that when you arrive at 1 or N you are basically stuck. You can do nothing more than stay in that state forever.

This function, given:

* a **starting point** (e.g. 3)
* the **first possible value** (e.g. 0)
* and **the last possible value** (e.g. 5)
* the **number of steps** (e.g. 10000)

Gives you the final state:

|  |
| --- |
| def gamblersruinchain(start,first,last,n):  for k in range(n):  if start==first or start==last:  start = start  else:  step = np.random.choice([-1,1],p=[0.5,0.5])  start = start + step  return start |

Now, before trying this function, let’s consider a more interesting situation.

Let’s say we start from state 3. What is the probability of ending up in state 5 after 2 steps?

Well, it is the probability of going from state 3 to state 4 and then going from state 4 to state 5:

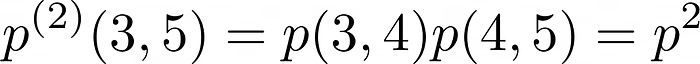


Image made by me using [LaTeX](https://latex2png.com/)

**In our case it would just be 0.25.**

If now we ask this equation:

Let’s say we start from state 3. What is the probability of ending up in state 1 after 2 steps?

Again, it is the probability of going from state 3 to state 2 and then going from state 2 to state 1:



Image made by me using [LaTeX](https://latex2png.com/)

**And, again, in our case it would just be 0.25.**

The only other option would be to go from state 3 to state 3 after two steps. We can compute this probability in a very easy way. As the total probability has to be 1, it is just:

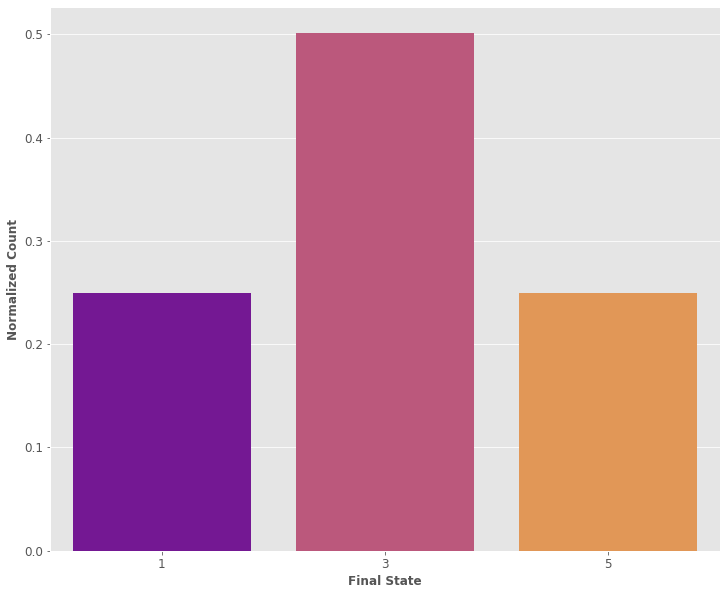


Image made by me using [LaTeX](https://latex2png.com/)

And if p=0.5 it is again 0.5

Again, the idea of probability is that if we repeat an experiment infinite times we should be able to verify the occurrences suggested by the probability values.

|  |
| --- |
| state\_list = []  for i in range(100000):  state\_list.append(gamblersruinchain(3,0,5,2))  data\_state = pd.DataFrame({'Final State':state\_list})  data\_occ = pd.DataFrame(data\_state.value\_counts('Final State')).rename(columns={0:'Count'})  data\_occ['Count'] = data\_occ['Count']/100000  sns.barplot(x=data\_occ.index,y=data\_occ['Count'],palette='plasma')  plt.ylabel('Normalized Count') |



**3. Custom Markov Chain**

The previous models are well known and used as introductory example of Markov Chains. Let’s try to be creative and build a whole new non existing model like the one **in the following picture.**

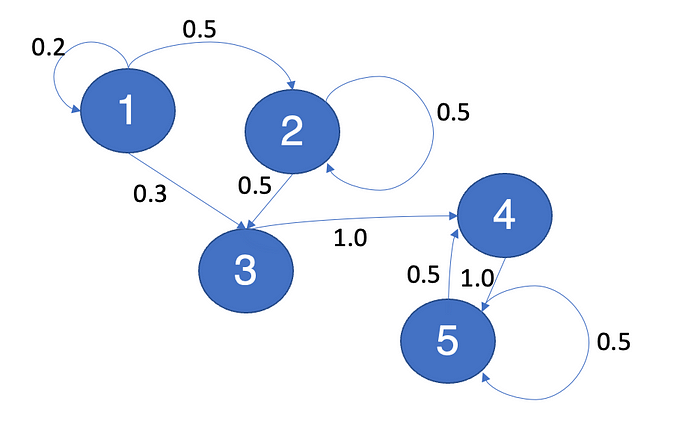


Image made by me using Power Point

I am a terrible drawer but the model itself is simple.

When you see an arrow between two nodes (let’s say A and B) it means that you can go to node B starting from node A with a certain probability that is written in black.

*For example it is possible to go from state A to state B with probability 0.5*

An important concept is that the model can be summarized using the transition matrix, that explains everything that can happen in your Markov chain. This is the **transition matrix** of our model:

|  |
| --- |
| trans\_matrix=[  [0.2,0.5,0.3,0,0],  [0,0.5,0.5,0,0],  [0,0,0,1.0,0],  [0,0,0,0,1],  [0,0,0,0.5,0.5]  ] |

If you look at the model closely you can see something very particular. Let’s say you jump from state 2 to state 3. Can you ever go back to state 2? The answer is no.

The same is valid for state 3 and state 1. State 1, 2 and 3 are thus defined **transient states.**

On the other hand, if you start from state 4, it is always possible that, at a certain time, you will go back to state 4. The same is valid for state 5. These states are known as **recurrent states.**

Let’s make some experiment so that we can properly understand this concept.

Intuitively, we can see that the probability of not coming back to state 2 starting from state 2 tends to 0 as the number of steps goes to infinity.

In fact, the probability that, starting from state 2 we find ourselves in state 2 after N step is the following:

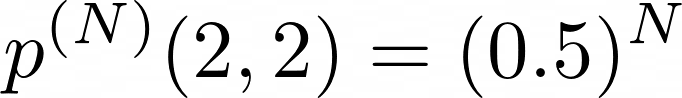
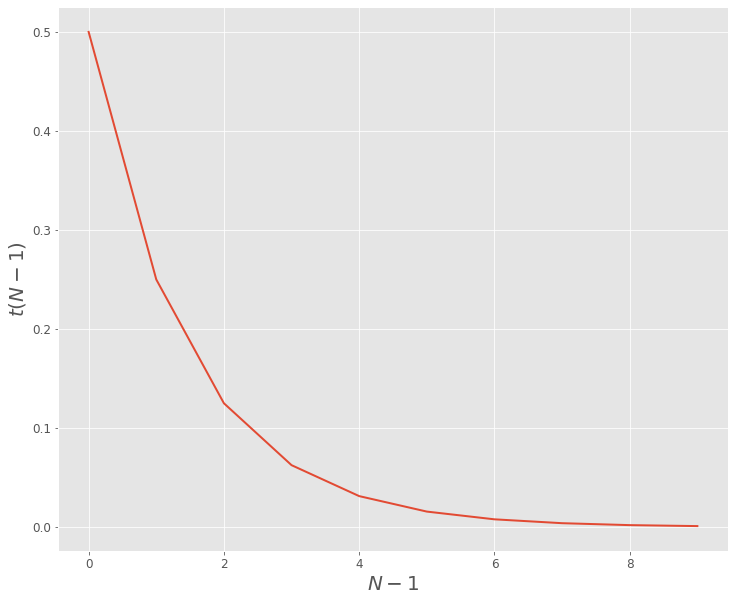


Image made by me using [LaTeX](https://latex2png.com/)

In fact, if we go from state 2 to state 3 it is impossible that we go back to state 2. Let’s define this theoretical function as t(N), and plot it:

|  |
| --- |
| def t(N):  step = np.arange(1,N+1,1)  y = []  for s in step:  v = 0.5\*\*s  y.append(v)  return y |

|  |
| --- |
| plt.plot(t(10))  plt.ylabel(r'$t(N-1)$',fontsize=20)  plt.xlabel(r'$N-1$',fontsize=20) |



Now, let’s use the Markov Chain and see if we verify the same results.

We start from state 2 and we verify after N step the probability of being in state 2. **The probability, in this case, is just the ratio between the number of 2 in the final state and the number of occurrences.** To be consistent, the number of occurrences needs to tend to infinity. Let’s consider 1000 tests.

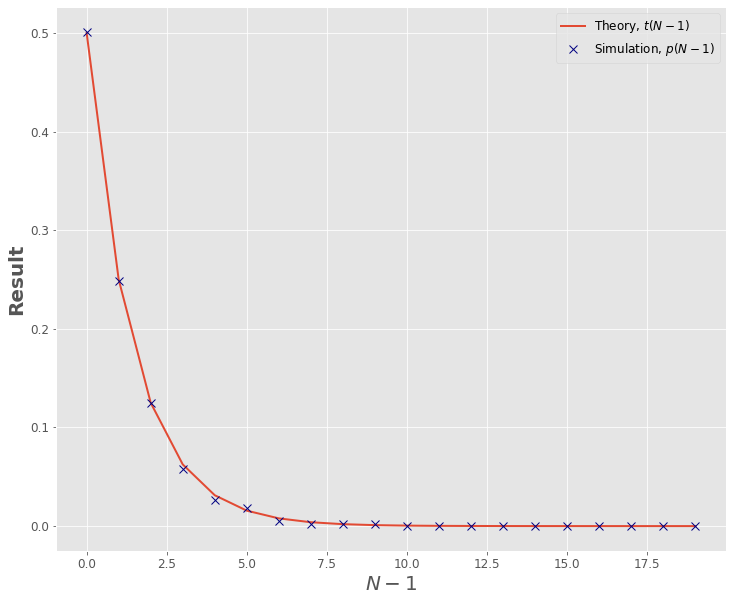
This is the function we are going to use:

|  |
| --- |
| def prob(N):  states = np.arange(1,6,1)  steps = np.arange(1,N+1,1)  n=1000  state\_collection = []  for k in range(n):  start = 2  for i in range(N):  start = np.random.choice(states,p=trans\_matrix[start-1])  if start==2:  state\_collection.append(1)  else:  state\_collection.append(0)  state\_collection = np.array(state\_collection)  return state\_collection.sum()/n |

Let’s use this function for various N and call this p(N):

|  |
| --- |
| def p(N):  step = np.arange(1,N+1,1)  y = []  for s in step:  v = prob(s)  y.append(v)  return y |

|  |
| --- |
| p\_20 = p(20)  plt.plot(t(20),label=r'Theory, $t(N-1)$')  plt.plot(p\_20,'x',label=r'Simulation, $p(N-1)$',color='navy')  plt.ylabel(r'Result',fontsize=20)  plt.xlabel(r'$N-1$',fontsize=20)  plt.legend() |



As it is possible to see, we have used the **transition matrix** to do this simulation. We can use the transition matrix to evaluate all the properties of the Markov Chain we are considering.

## 4. Conclusions

In this notebook we have seen very well known models as the **Random Walks** and the **Gambler’s ruin chain**. Then we created our own brand new model and we played a little bit with it, discovering important concepts like **the one of the transition matrix, recurrent states and transient states.** Most importantly, we have seen how to verify these concepts in a very simple way using Python and very well known libraries.