

# [220] Randomness

Meena Syamkumar  
Mike Doescher

# Announcements

- P13
  - Due Wednesday Dec 9<sup>th</sup>
  - No late days allowed
  - No resubmissions allowed (exempt: 0 score students)
- Peer mentor interest form (coming soon)
- Want more?
  - Direct follow up course: CS 320
  - Computer Sciences: CS 200, 300, 400
- Office Hours
  - Last day of TA office hours Wednesday Dec 9<sup>th</sup>.
  - Instructor hours information – coming soon

# Final exam

- Recommended prep
  - make sure you understand all the worksheet problems
  - review the readings, slides, **lecture demo code**
  - review everything you got wrong on the midterms
  - review the code you wrote for the projects
  - prepare a note sheet (despite open material!)
- Live review session on Wednesday Dec 9<sup>th</sup>
  - optional Q/A BBC session
  - attend any session

# Course evaluations

- We value student feedback greatly
- Please bring a smile to your instructors' face by spending a few minutes to fill out evals 😊
  - CS220 SEC 001:  
<https://aefis.wisc.edu/index.cfm/page/AefisCourseSection.home?courseSectionid=645838>
  - CS220 SEC 002:  
<https://aefis.wisc.edu/index.cfm/page/AefisCourseSection.home?courseSectionid=645837>
  - CS220 SEC 003: Link to be updated soon
  - CS319 SEC 001:  
<https://aefis.wisc.edu/index.cfm/page/AefisCourseSection.home?courseSectionid=661619>
  - CS319 SEC 002:  
<https://aefis.wisc.edu/index.cfm/page/AefisCourseSection.home?courseSectionid=661618>
  - CS319 SEC 003: Link to be updated soon

# Which series was randomly generated?

## Which did I pick by hand?

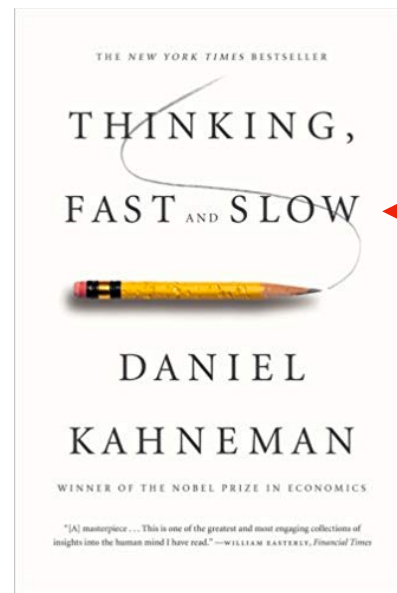
1



2

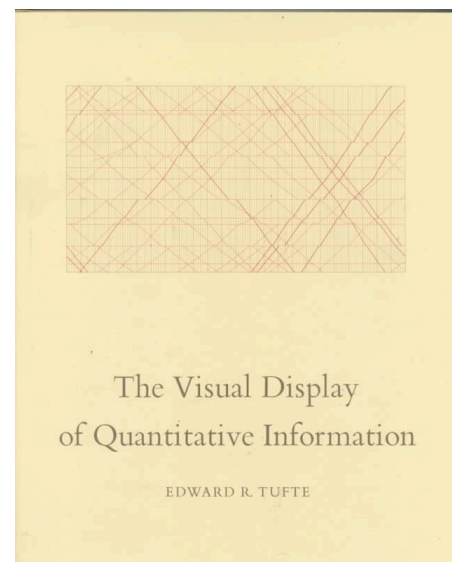


# Recommended winter reading

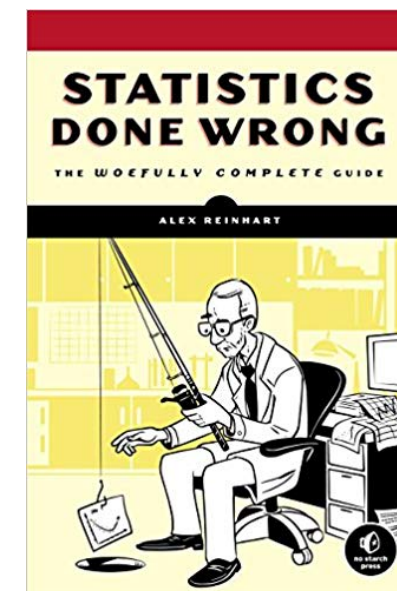


**Thinking, Fast and Slow**  
by Daniel Kahneman

*Misconceptions of chance.* People expect that a sequence of events generated by a random process will represent the essential characteristics of that process even when the sequence is short. In considering tosses of a coin for heads or tails, for example, people regard the sequence H-T-H-T-T-H to be more likely than the sequence H-H-H-T-T-T, which does not appear random, and also more likely than the sequence H-H-H-H-T-H, which does not represent the fairness of the coin.<sup>7</sup> Thus,

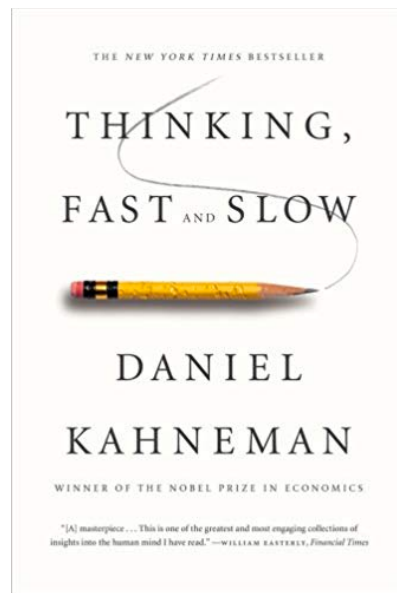


**The Visual Display of Quantitative Information**  
by Edward R. Tufte

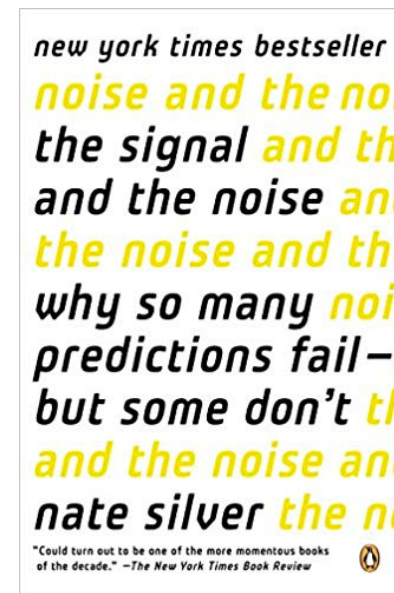


**Statistics Done Wrong**  
by Alex Reinhart

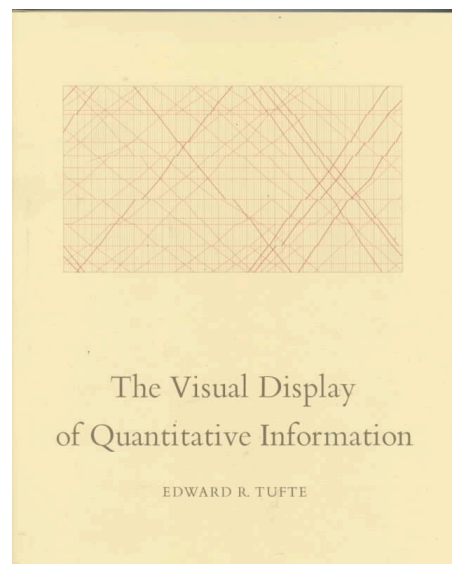
# Recommended winter reading



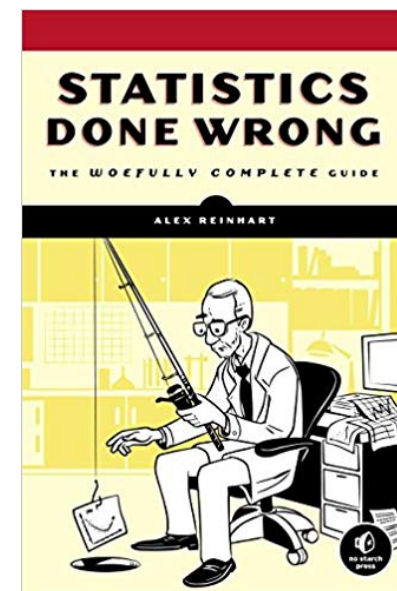
**Thinking, Fast and Slow**  
by Daniel Kahneman



**The Signal and the Noise**  
by Nate Silver



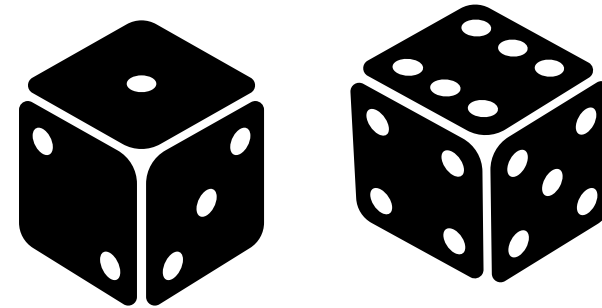
**The Visual Display of Quantitative Information**  
by Edward R. Tufte



**Statistics Done Wrong**  
by Alex Reinhart

# Why Randomize?

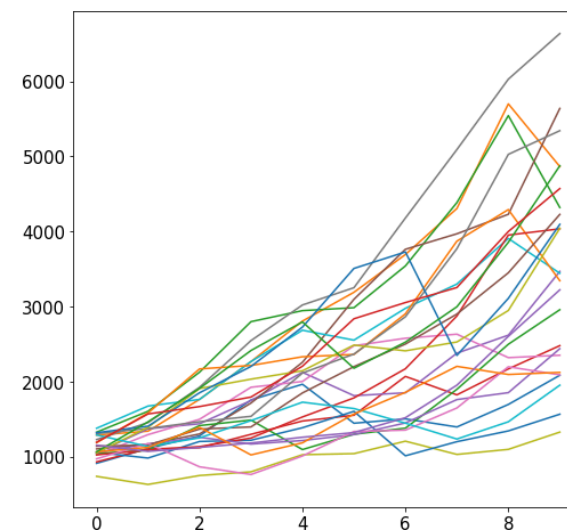
Games



Security



Simulation



our focus



# Outline

choice()

bugs and seeding

significance

histograms

normal()

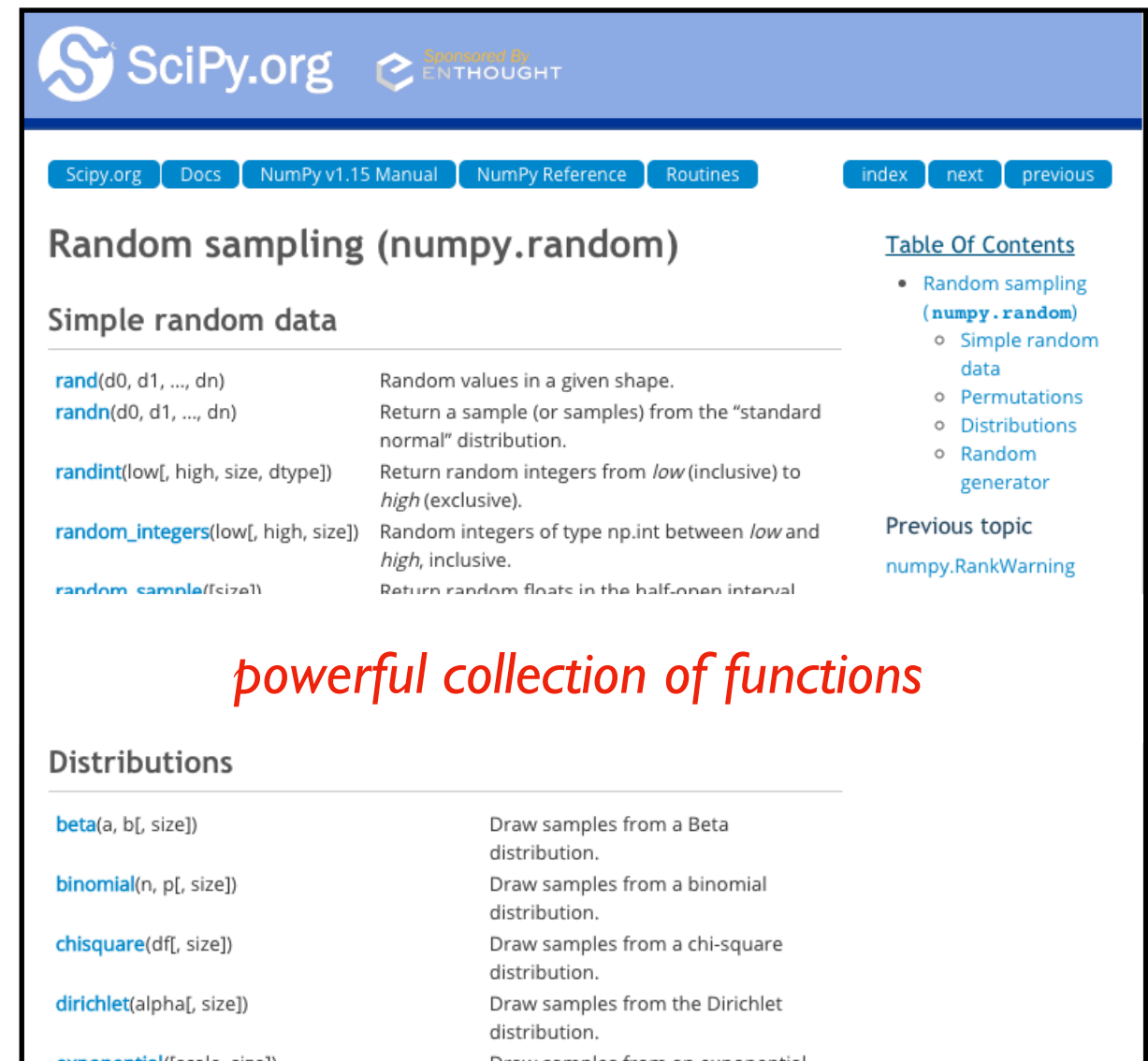
# New Functions Today

## `numpy.random`:

- powerful collection of functions
- **Choice**

## `Series.plot.hist`:

- similar to bar plot
- visualize spread of random results



The screenshot shows the SciPy.org website with the 'Random sampling (numpy.random)' section. The page includes a navigation bar with links to 'SciPy.org', 'Docs', 'NumPy v1.15 Manual', 'NumPy Reference', and 'Routines'. A 'Table Of Contents' on the right lists 'Random sampling (numpy.random)' with sub-items: 'Simple random data', 'Permutations', 'Distributions', and 'Random generator'. The 'Simple random data' section lists functions: `rand`, `randn`, `randint`, `random_integers`, and `random_sample`. The 'Distributions' section lists functions: `beta`, `binomial`, `chisquare`, `dirichlet`, and `exponential`. A red text overlay 'powerful collection of functions' is positioned over the 'Simple random data' section.

SciPy.org

Random sampling (numpy.random)

Simple random data

<code>rand(d0, d1, ..., dn)</code>	Random values in a given shape.
<code>randn(d0, d1, ..., dn)</code>	Return a sample (or samples) from the "standard normal" distribution.
<code>randint(low[, high, size, dtype])</code>	Return random integers from <i>low</i> (inclusive) to <i>high</i> (exclusive).
<code>random_integers(low[, high, size])</code>	Random integers of type np.int between <i>low</i> and <i>high</i> , inclusive.
<code>random_sample(size)</code>	Return random floats in the half-open interval

Distributions

<code>beta(a, b[, size])</code>	Draw samples from a Beta distribution.
<code>binomial(n, p[, size])</code>	Draw samples from a binomial distribution.
<code>chisquare(df[, size])</code>	Draw samples from a chi-square distribution.
<code>dirichlet(alpha[, size])</code>	Draw samples from the Dirichlet distribution.
<code>exponential(scale, size)</code>	Draw samples from an exponential

powerful collection of functions

# choice

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

```
result = choice([<choice1, choice2, ...])
```



**list of things to  
randomly choose from**

# choice

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

```
result = choice(["rock", "paper", "scissors"])
```

**list of things to  
randomly choose from**



# choice

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

```
result = choice(["rock", "paper", "scissors"])  
print(result)
```



## Output:

scissors

# choice


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

```
result = choice(["rock", "paper", "scissors"])  
print(result)
```

```
result = choice(["rock", "paper", "scissors"])  
print(result)
```

## Output:

scissors  
rock



each time choice is  
called, a value is randomly  
selected (will vary run to run)

# choice

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

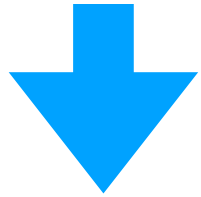
```
choice(["rock", "paper", "scissors"], size=5)
```

**for simulation, we'll often want  
to compute many random results**

# choice

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

```
choice(["rock", "paper", "scissors"], size=5)
```



```
array(['rock', 'scissors', 'paper', 'rock', 'paper'], dtype='<U8')
```

it's list-like



# Random values and Pandas

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

```
# random Series
```

```
Series(choice(["rock", "paper", "scissors"], size=5))
```

```
0      rock
1      rock
2  scissors
3     paper
4  scissors
dtype: object
```

# Random values and Pandas

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

```
# random Series
```

```
DataFrame(choice(["rock", "paper", "scissors"],  
                 size=(5,2)))
```

↓ →

	0	1
0	paper	rock
1	scissors	rock
2	rock	rock
3	scissors	paper
4	rock	scissors

# Demo: exploring bias

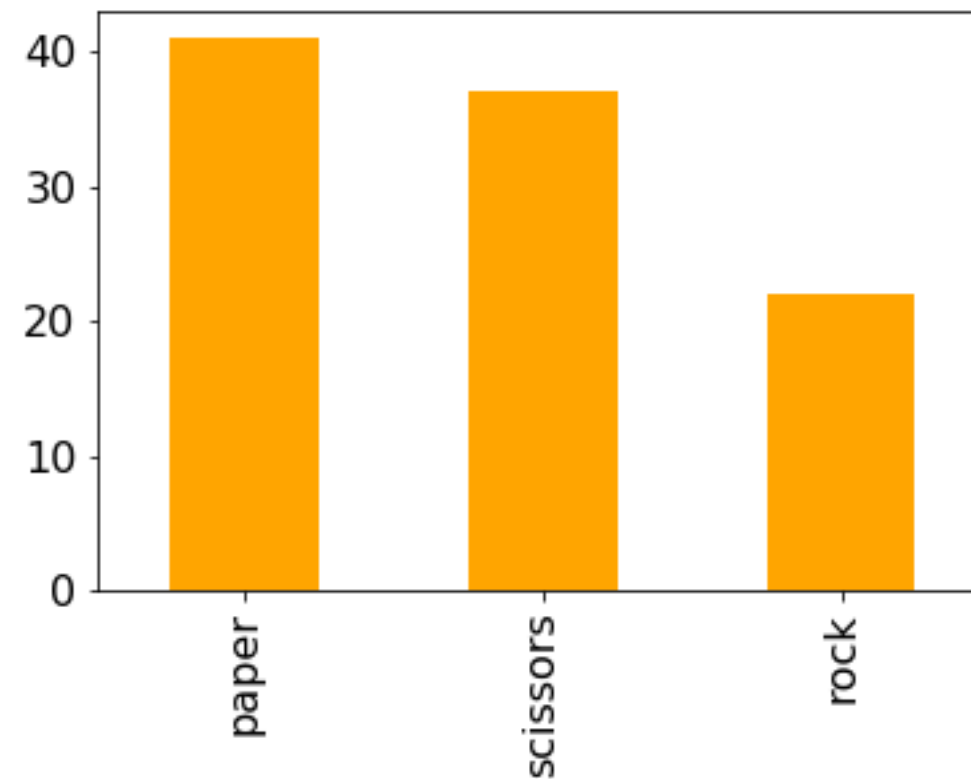
```
choice([ "rock", "paper", "scissors" ])
```

**Question 1: how can we make sure the randomization isn't biased?**

# Demo: exploring bias

```
choice(["rock", "paper", "scissors"])
```

**Question 1: how can we make sure the randomization isn't biased?**

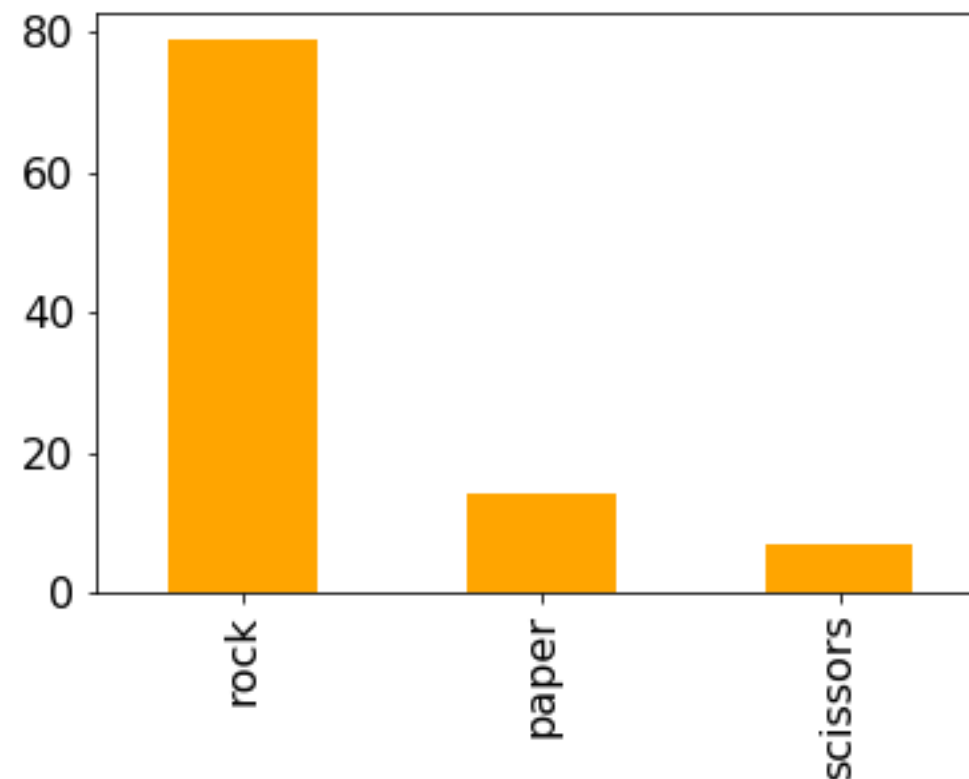


# Demo: exploring bias

```
choice(["rock", "paper", "scissors"])
```

**Question 1: how can we make sure the randomization isn't biased?**

**Question 2: how can we make it biased (if we want it to be)?**



**p=[...]**

# Random Strings vs. Random Ints

```
from numpy.random import choice, normal

# random string: rock, paper, or scissors
choice(["rock", "paper", "scissors"])

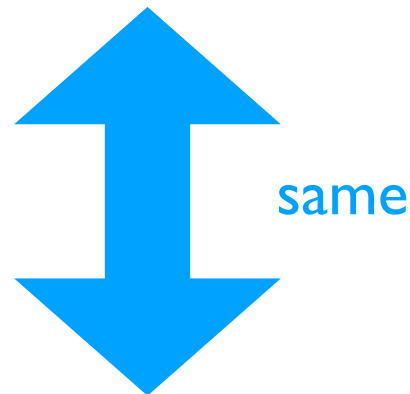
# random int: 0, 1, or 2
choice([0, 1, 2])
```

# Random Strings vs. Random Ints

```
from numpy.random import choice, normal
```

```
# random string: rock, paper, or scissors  
choice(["rock", "paper", "scissors"])
```

```
# random int: 0, 1, or 2  
choice([0, 1, 2])
```



```
# random int (approach 2): 0, 1, or 2  
choice(3)
```

random non-negative int  
that is **less than 3**

# Outline

choice()

bugs and seeding

significance

histograms

normal()



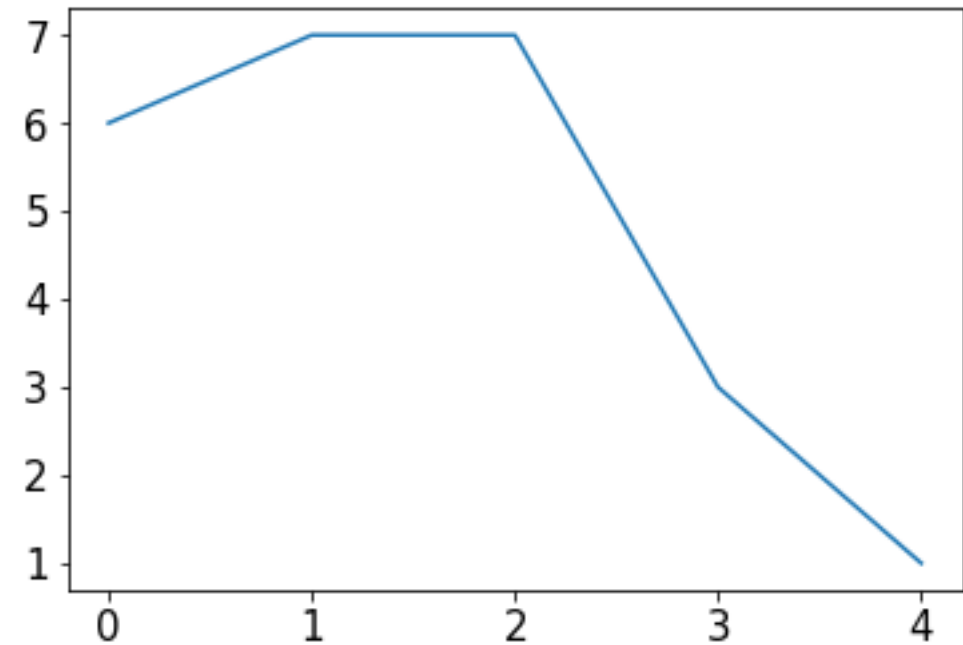
# Example: change over time

```
s = Series(choice(10, size=5))
```

0	6
1	7
2	7
3	3
4	1

dtype: int64

```
s.plot.line()
```

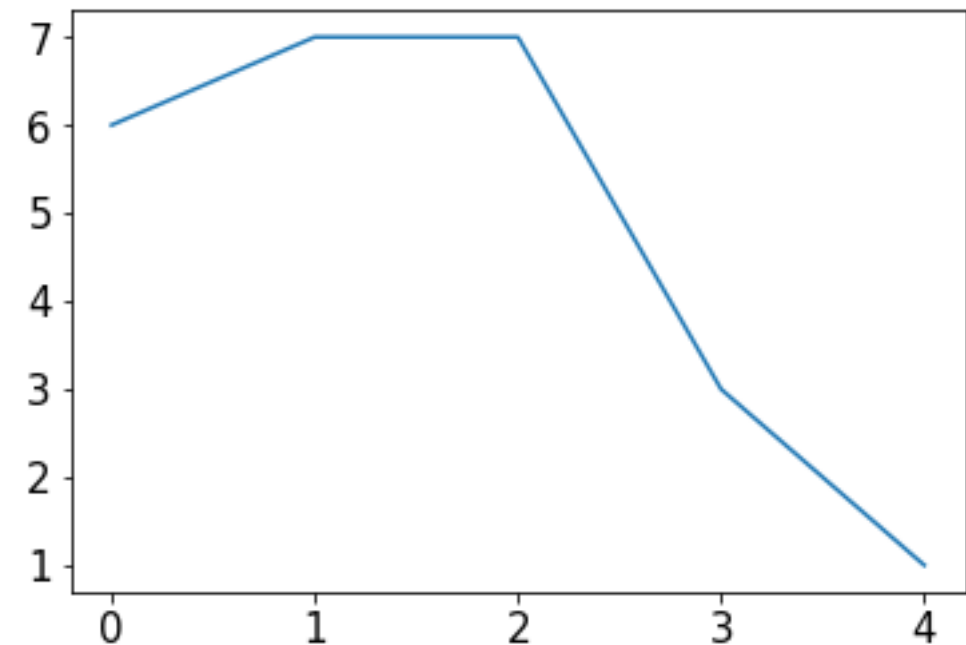


# Example: change over time

```
s = Series(choice(10, size=5))
```

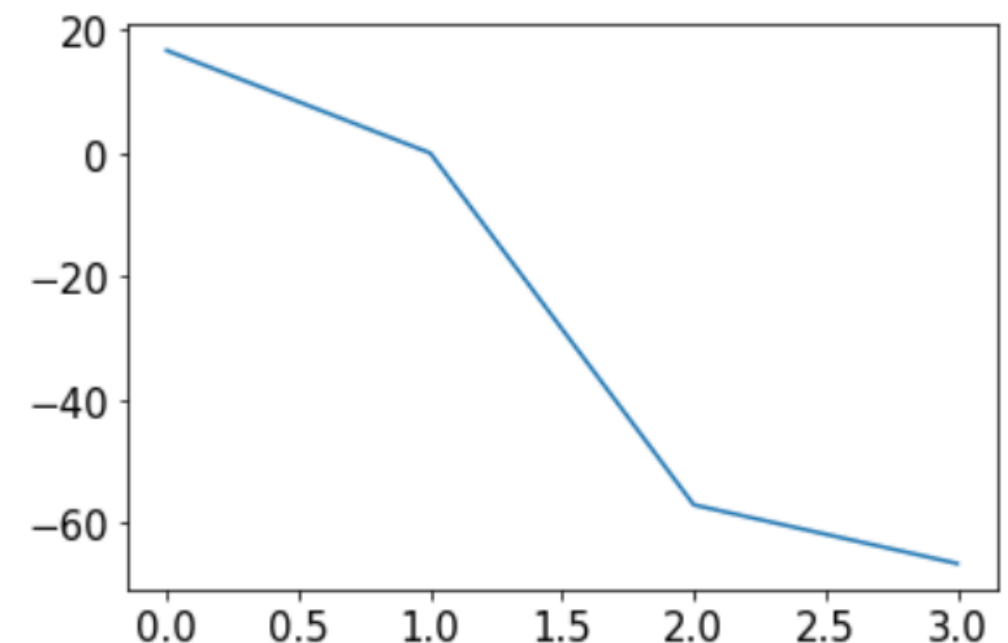
0	6
1	7
2	7
3	3
4	1
dtype: int64	

```
s.plot.line()
```



```
percents = []  
for i in range(1, len(s)):  
    diff = 100 * (s[i] / s[i-1] - 1)  
    percents.append(diff)  
Series(percents).plot.line()
```

*what are we computing for diff?*

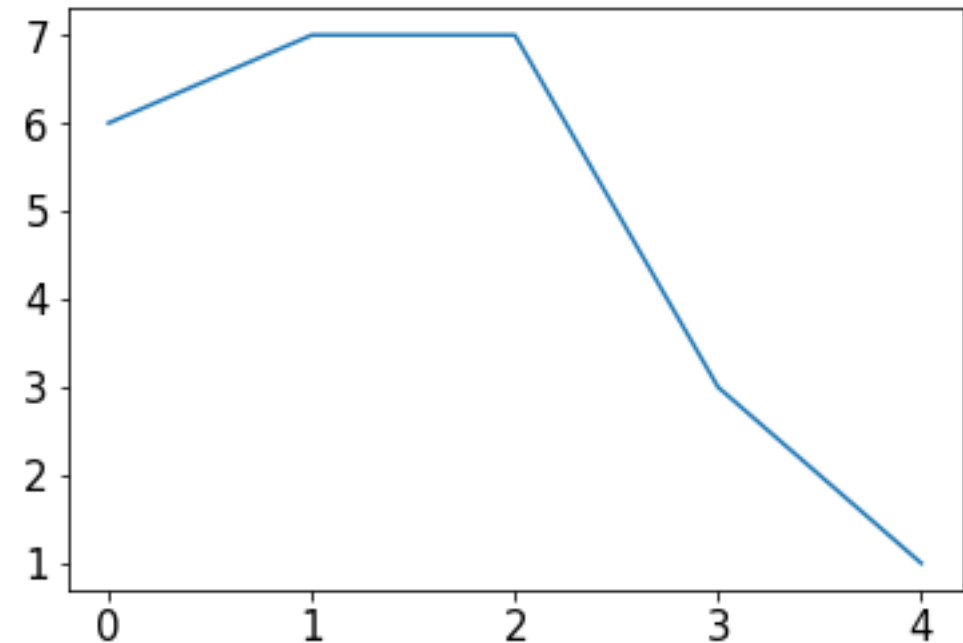


# Example: change over time

```
s = Series(choice(10, size=5))
```

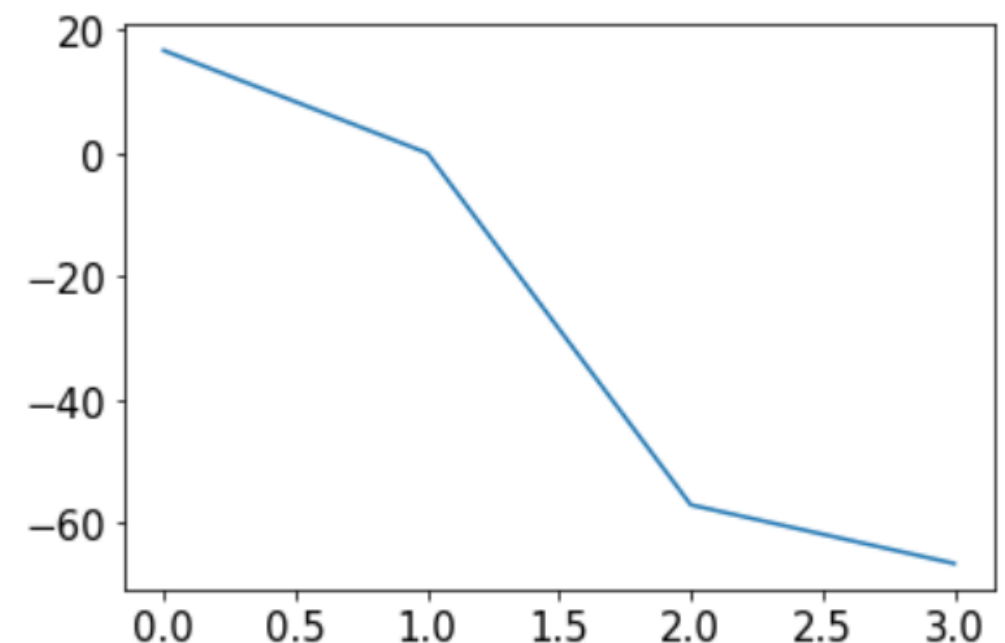
0	6
1	7
2	7
3	3
4	1
dtype: int64	

```
s.plot.line()
```



```
percents = []  
for i in range(1, len(s)):  
    diff = 100 * (s[i] / s[i-1] - 1)  
    percents.append(diff)  
Series(percents).plot.line()
```

*can you identify the bug in the code?*



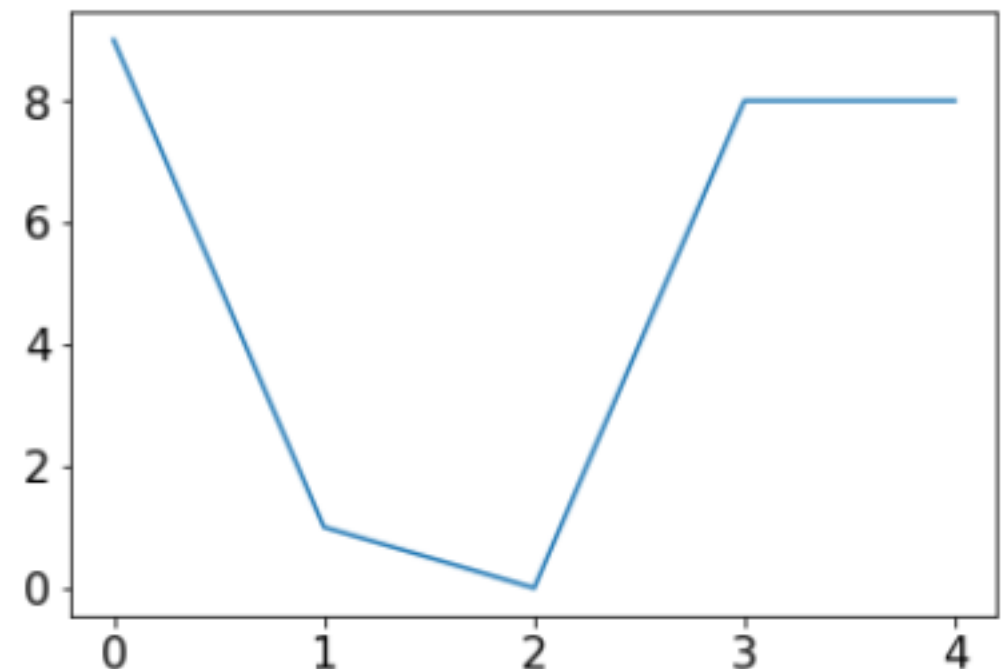
# Example: change over time

```
s = Series(choice(10, size=5))
```

0	9
1	1
2	0
3	8
4	8

dtype: int64

```
s.plot.line()
```



```
percents = []  
for i in range(1, len(s)):  
    diff = 100 * (s[i] / s[i-1] - 1)  
    percents.append(diff)  
Series(percents).plot.line()
```

*can you identify the bug in the code?*

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/  
python3.7/site-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero encountered in long_scalars  
This is separate from the ipykernel package so we can  
avoid doing imports until
```

# Not all bugs are equal!

## scary bugs

non-deterministic



Igor Siwanowicz

## "nice" bugs

deterministic (reproducible)



# Not all bugs are equal!

## scary bugs

non-deterministic  
system related  
randomness



Igor Siwanowicz

## "nice" bugs

deterministic (reproducible)



# Not all bugs are equal!

## scary bugs

non-deterministic  
system related  
randomness

large data

semantic



Igor Siwanowicz

## "nice" bugs

deterministic (reproducible)

small data

syntax



runtime

# Not all bugs are equal!

## scary bugs

non-deterministic  
system related  
randomness

large data

semantic

????

runtime



Igor Siwanowicz

## "nice" bugs

deterministic (reproducible)

small data

syntax





# Not all bugs are equal!

scary bugs

"nice" bugs

non-deterministic  
system related  
randomness

seeding

deterministic (reproducible)

large data

small data

semantic

assert

syntax

runtime



Igor Siwanowicz



# Pseudorandom Generators


"Random" generators are really just *pseudorandom*



684	559	629	192	835	...
37	235	908	72	767	...
168	527	493	584	534	...
874	664	249	643	952	...

# Pseudorandom Generators

"Random" generators are really just *pseudorandom*



684	559	629	192	835	...
-----	-----	-----	-----	-----	-----

```
In [39]: 1 choice(1000, size=3)|
```

```
Out[39]: array([684, 559, 629])
```

168	527	493	584	534	...
-----	-----	-----	-----	-----	-----

874	664	249	643	952	...
-----	-----	-----	-----	-----	-----

# Pseudorandom Generators

"Random" generators are really just *pseudorandom*



684	559	629	192	835	...
37	235	908	72	767	...
168	527	493	584	534	...
874	664	249	643	952	...

# Pseudorandom Generators

"Random" generators are really just *pseudorandom*

seeds



**100:**

684 559 629 192 835 ...

**101:**

37 235 908 72 767 ...

**102:**

168 527 493 584 534 ...

...

874 664 249 643 952 ...

# Pseudorandom Generators

What if I told you that you can **choose** your track?

seeds



**100:**

684 559 629 192 835 ...

**101:**

37 235 908 72 767 ...

**102:**

168 527 493 584 534 ...

...

874 664 249 643 952 ...

# Seeding

What if I told you that you can **choose** your track?

```
In [2]: 1 np.random.seed(220)
        2 choice(1000, size = 3)
```

```
Out[2]: array([883, 732, 15])
```

```
In [3]: 1 np.random.seed(220)
        2 choice(1000, size = 3)
```

```
Out[3]: array([883, 732, 15])
```

```
In [4]: 1 np.random.seed(220)
        2 choice(1000, size = 3)
```

```
Out[4]: array([883, 732, 15])
```

# Seeding

Common approach for simulations:

1. seed using current time
2. print seed
3. use the seed for reproducing bugs, as necessary

In [28]:

```
1 import time
2 now = int(time.time())
3 print("seeding with", now)
4 np.random.seed(now)
5 choice(1000, size=3)
```

seeding with 1556673136

Out[28]: array([352, 734, 362])



# Outline

choice()

bugs and seeding

significance

histograms

normal()

# In a noisy world, what is noteworthy?



# Is this coin biased?



**51**



**49**

Call shenanigans?



**whoever has the coin cheated  
(it's not 50/50 heads/tails)**

a statistician might say we're  
trying to decide if the evidence  
that the coin isn't fair is  
**statistically significant**

# Is this coin biased?



**51**



**49**

Call shenanigans? No.

# Is this coin biased?



**51**



**49**

Call shenanigans? No.



**5**



**95**

Call shenanigans?

# Is this coin biased?



51



49

Call shenanigans? No.



5



95

Call shenanigans? Yes.

**Note:** there is a non-zero probability that a fair coin will do this, but the odds are slim

# Is this coin biased?



**51**



**49**

Call shenanigans? No.



**5**



**95**

Call shenanigans? Yes.



**55**



**45**

Call shenanigans?



Call shenanigans?

**55 million 45 million**



# Is this coin biased?



**51**



**49**

Call shenanigans? No.



**5**



**95**

Call shenanigans? Yes.



**55**



**45**

Call shenanigans? No.



Call shenanigans? Yes.

**55 million 45 million**



# Is this coin biased?



51



49

Call shenanigans? No.



5



95

Call shenanigans? Yes.

**large skew** is good evidence of shenanigans



55



45

Call shenanigans? No.



**55 million 45 million**



Call shenanigans? Yes.

small skew over **large samples** is good evidence

# Demo: CoinSim



**60**



**40**

Call shenanigans?

**Strategy:** simulate a fair coin

1. "flip" it 100 times using `numpy.random.choice`
2. count heads
3. repeat above 10K times

[50, 61, 51, 44, 39, 43, 51, 49, 49, 38, ...]

# Demo: CoinSim



**60**



**40**

Call shenanigans?

we got 10 more heads than we expect on average  
how common is this?

**Strategy:** simulate a fair coin

1. "flip" it 100 times using `numpy.random.choice`
2. count heads
3. repeat above 10K times

[50, 61, 51, 44, 39, 43, 51, 49, 49, 38, ...]

# Demo: CoinSim



**60**



**40**

Call shenanigans?

we got 10 more heads than we expect on average  
how common is this?

**Strategy:** simulate a fair coin

1. "flip" it 100 times using `numpy.random.choice`
2. count heads
3. repeat above 10K times

[50, 61, 51, 44, 39, 43, 51, 49, 49, 38, ...]

11 more

12 less

# Outline

choice()

bugs and seeding

significance

histograms

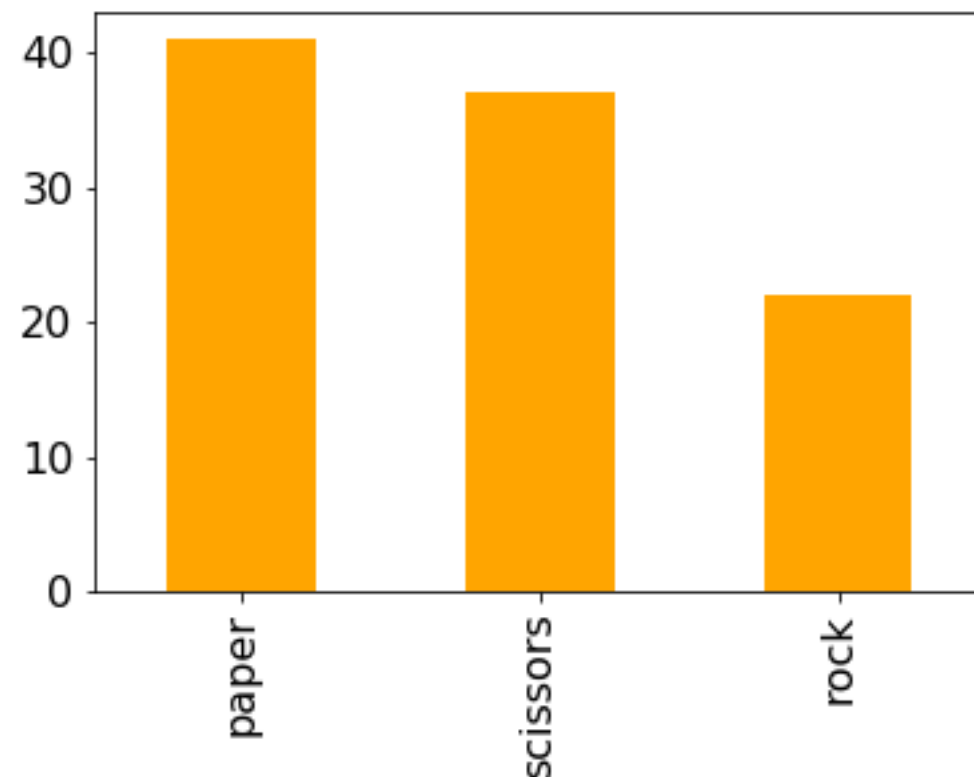
normal()

# Frequencies across categories

bars are a **good way** to view frequencies **across categories**

```
s = Series(["rock", "rock", "paper",  
           "scissors", "scissors", "scissors"])
```

```
s.value_counts().plot.bar(color="orange")
```

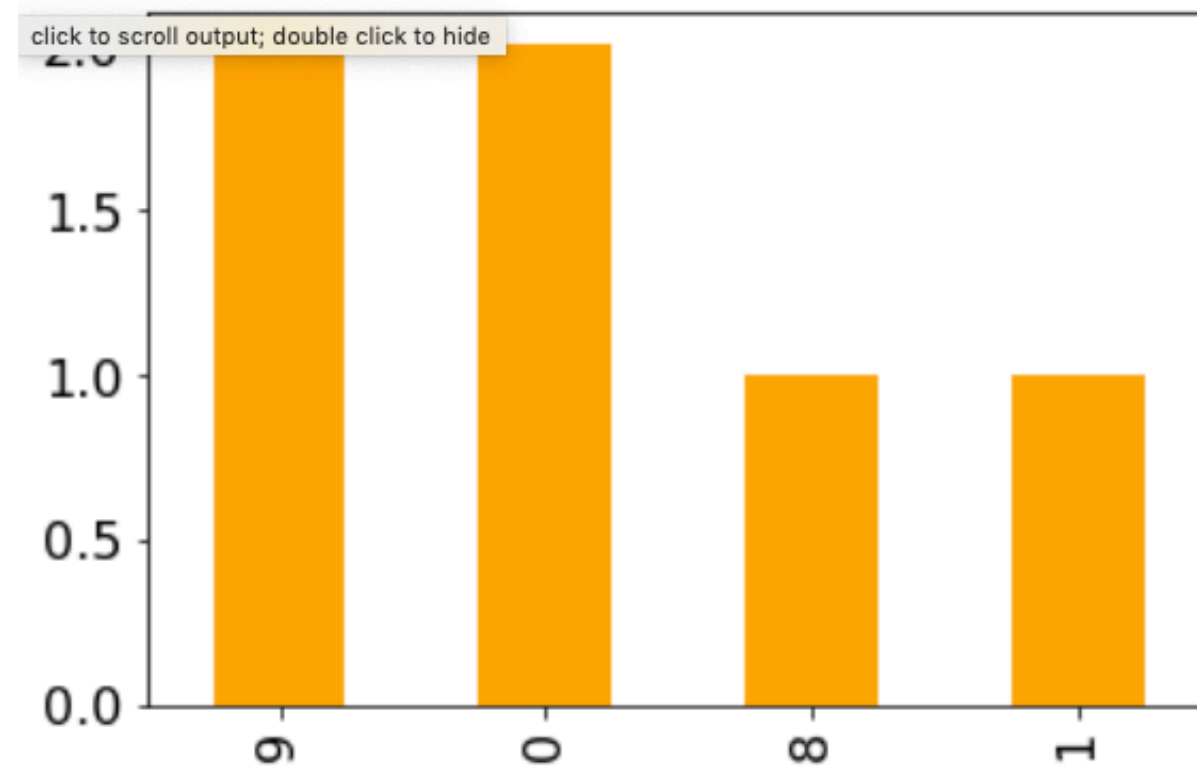


# Frequencies across numbers

bars are a **bad way** to view frequencies **across numbers**

```
s = Series([0, 0, 1, 8, 9, 9])
```

```
s.value_counts().plot.bar(color="orange")
```



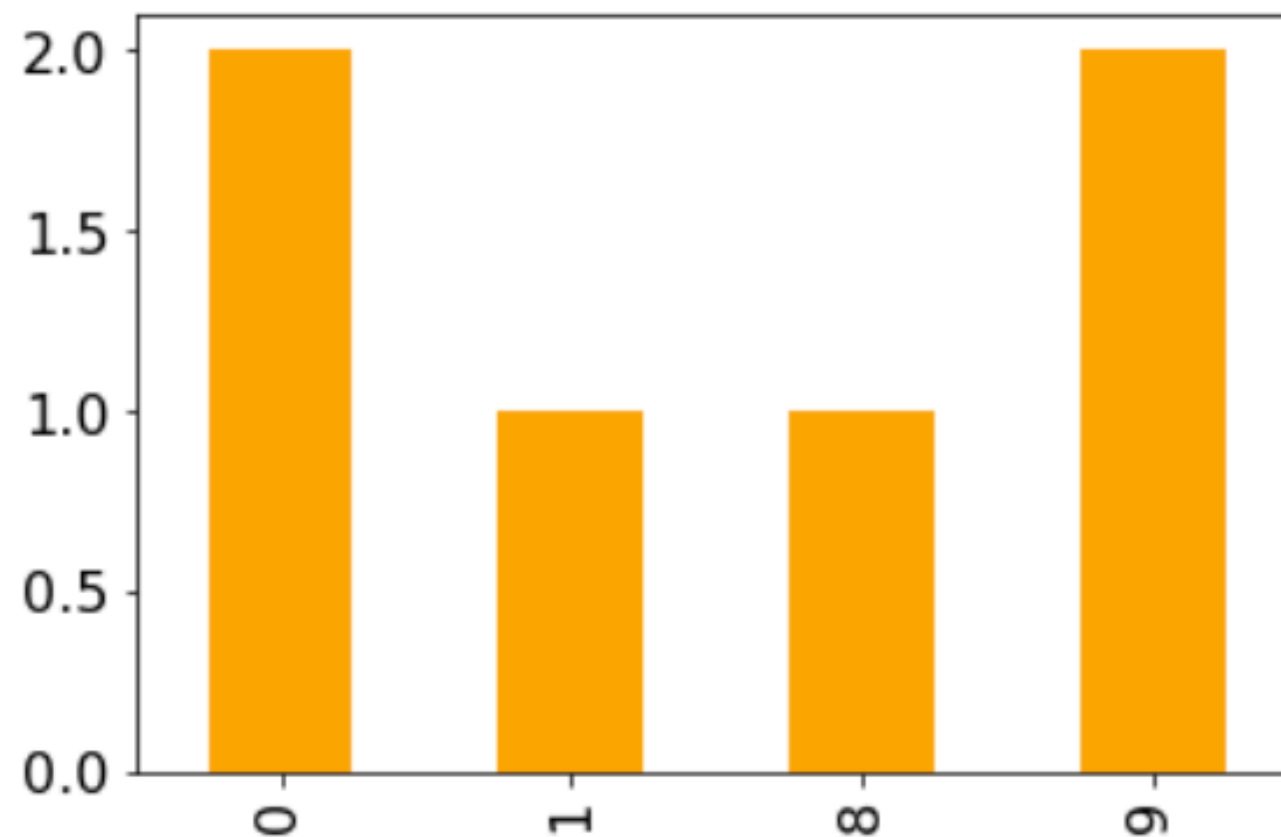
**numbers not ordered**

# Frequencies across numbers

bars are a **bad way** to view frequencies **across numbers**

```
s = Series([0, 0, 1, 8, 9, 9])
```

```
s.value_counts().sort_index().plot.bar(color="orange")
```



gap between 1 and 8 not obvious

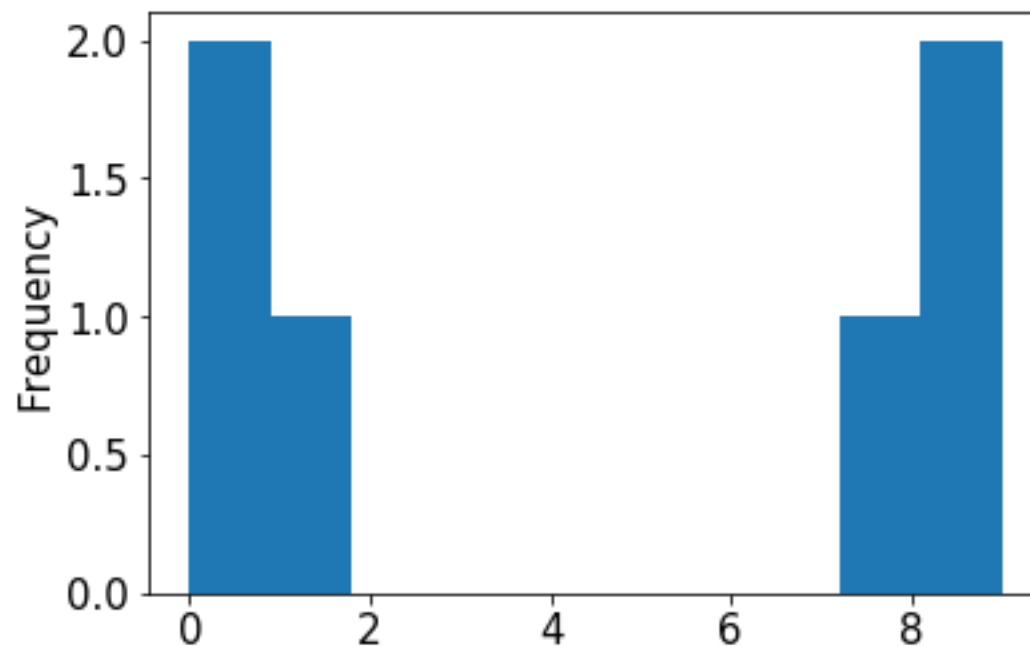


# Frequencies across numbers

bars are a **bad way** to view frequencies **across numbers**

```
s = Series([0, 0, 1, 8, 9, 9])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist()
```

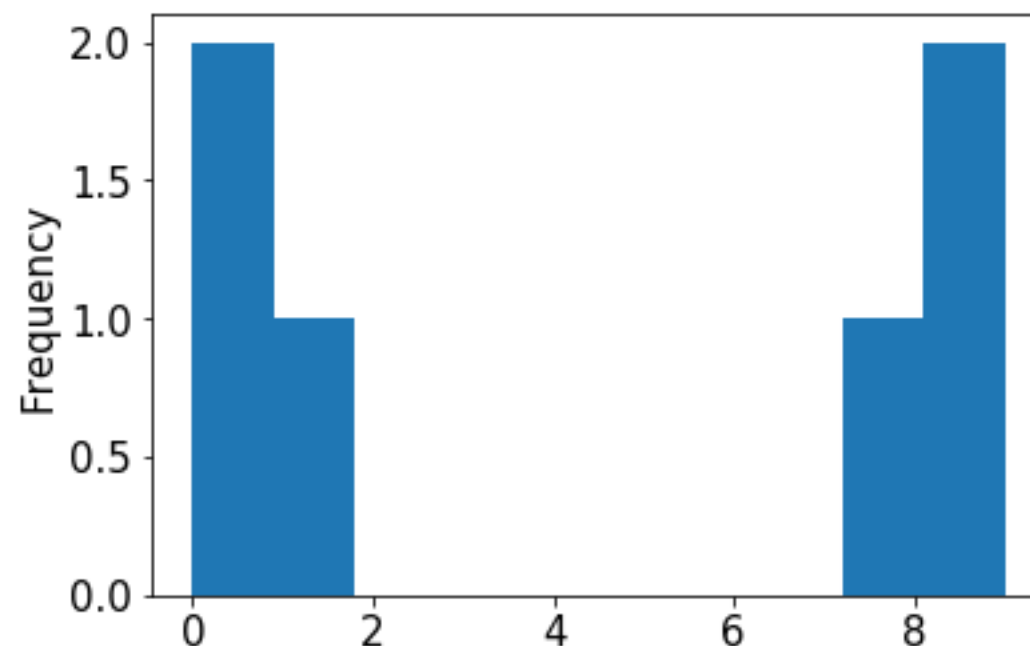


# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0, 0, 1, 8, 9, 9])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist()
```



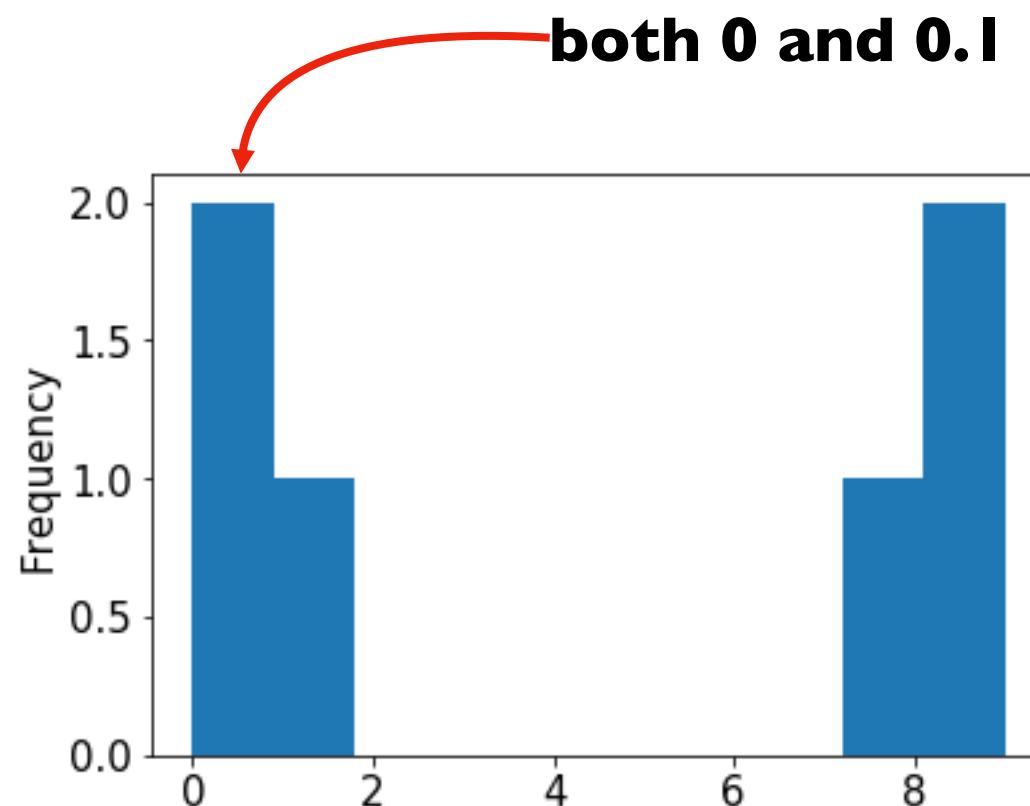
**this kind of plot is called a histogram**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist()
```



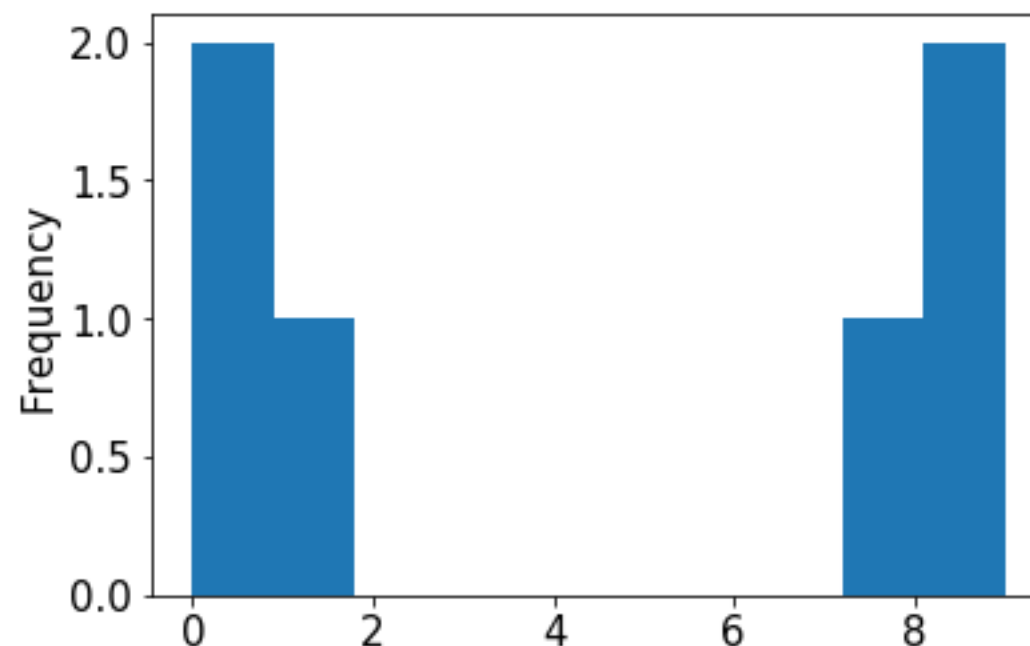
**a histogram "bins" nearby numbers to create discrete bars**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=10)
```



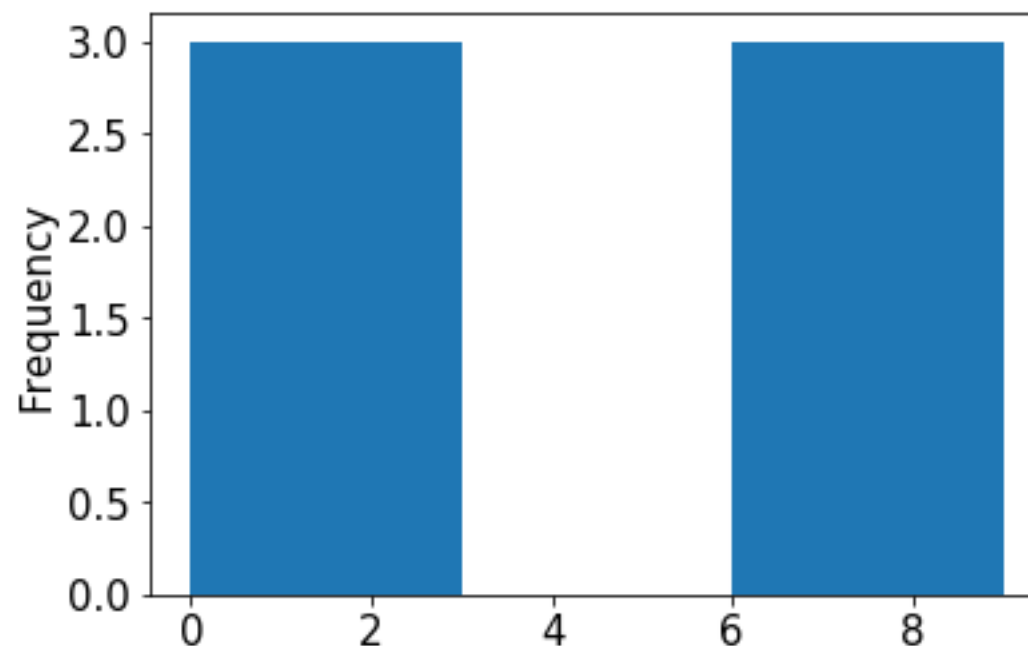
**we can control the number of bins**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=3)
```



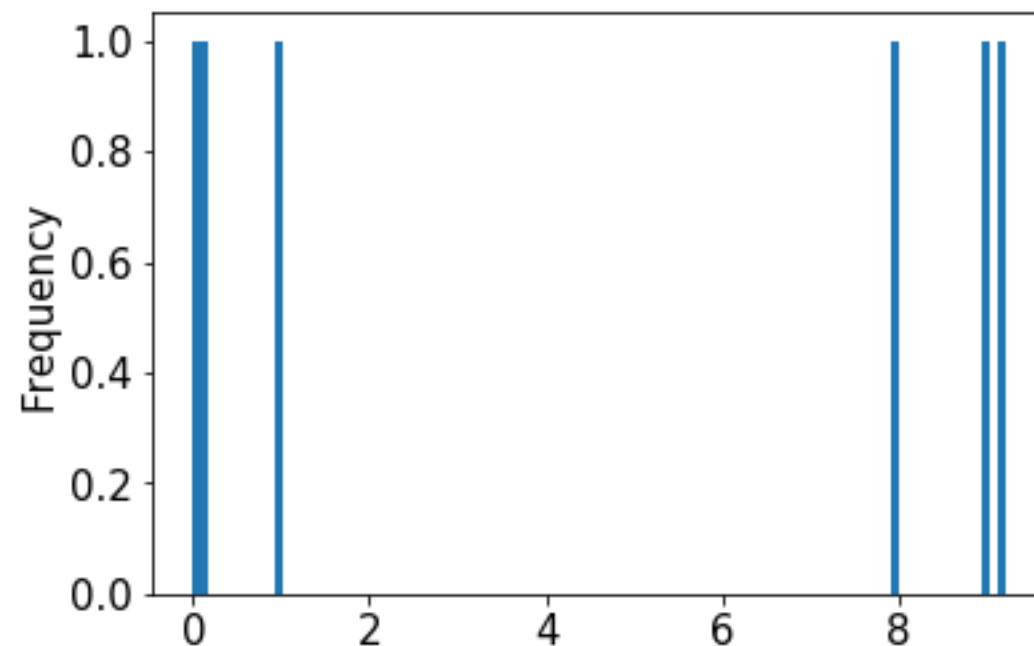
**too few bins provides too little detail**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=100)
```



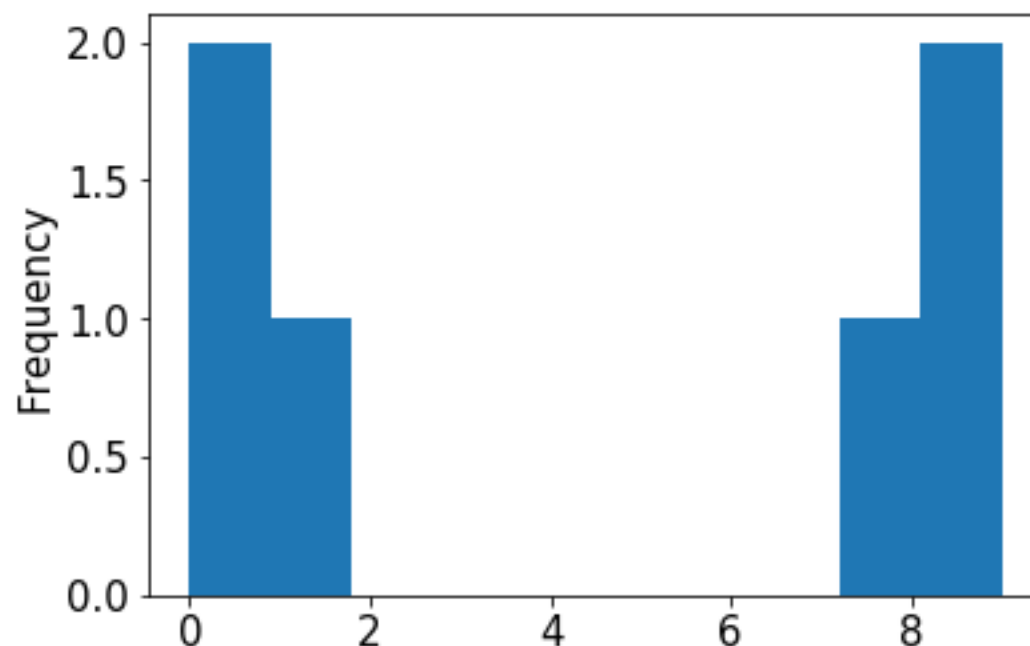
**too many bins provides too much detail (equally bad)**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=10)
```



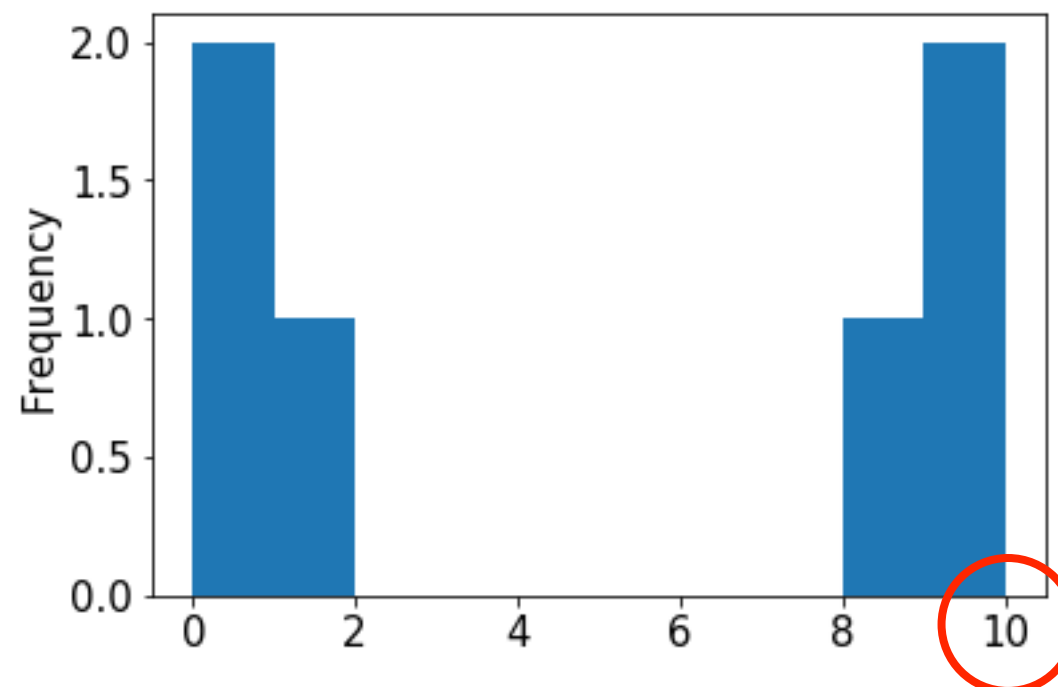
**pandas chooses the default bin boundaries**

# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=[0,1,2,3,4,5,6,7,8,9,10])
```



**we can override the defaults**

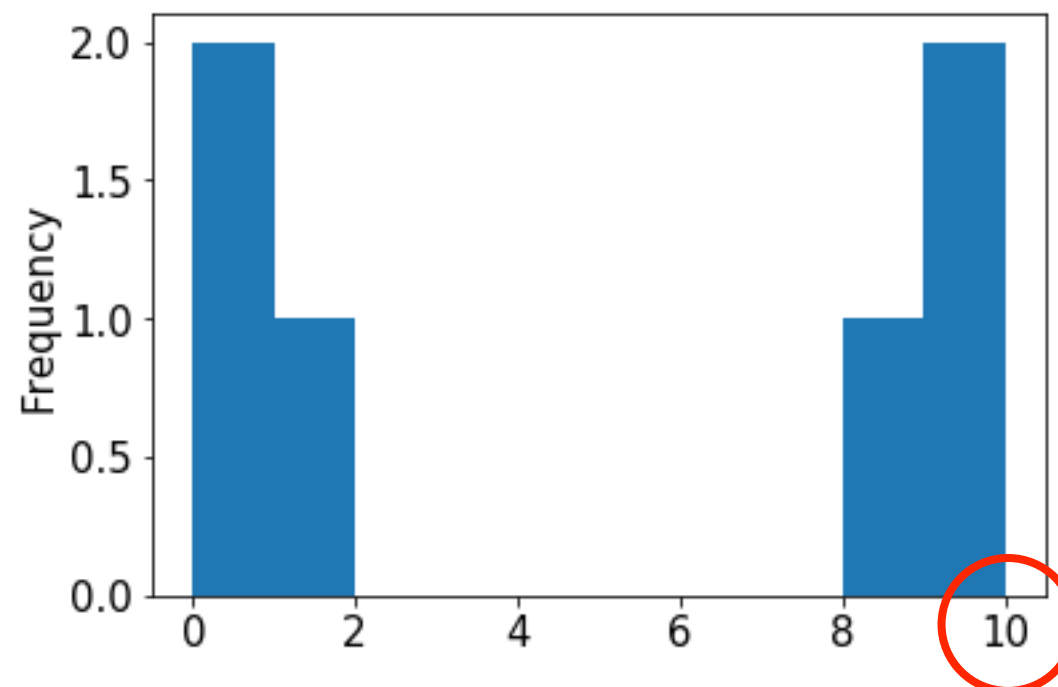


# Frequencies across numbers

histograms are a **good way** to view frequencies **across numbers**

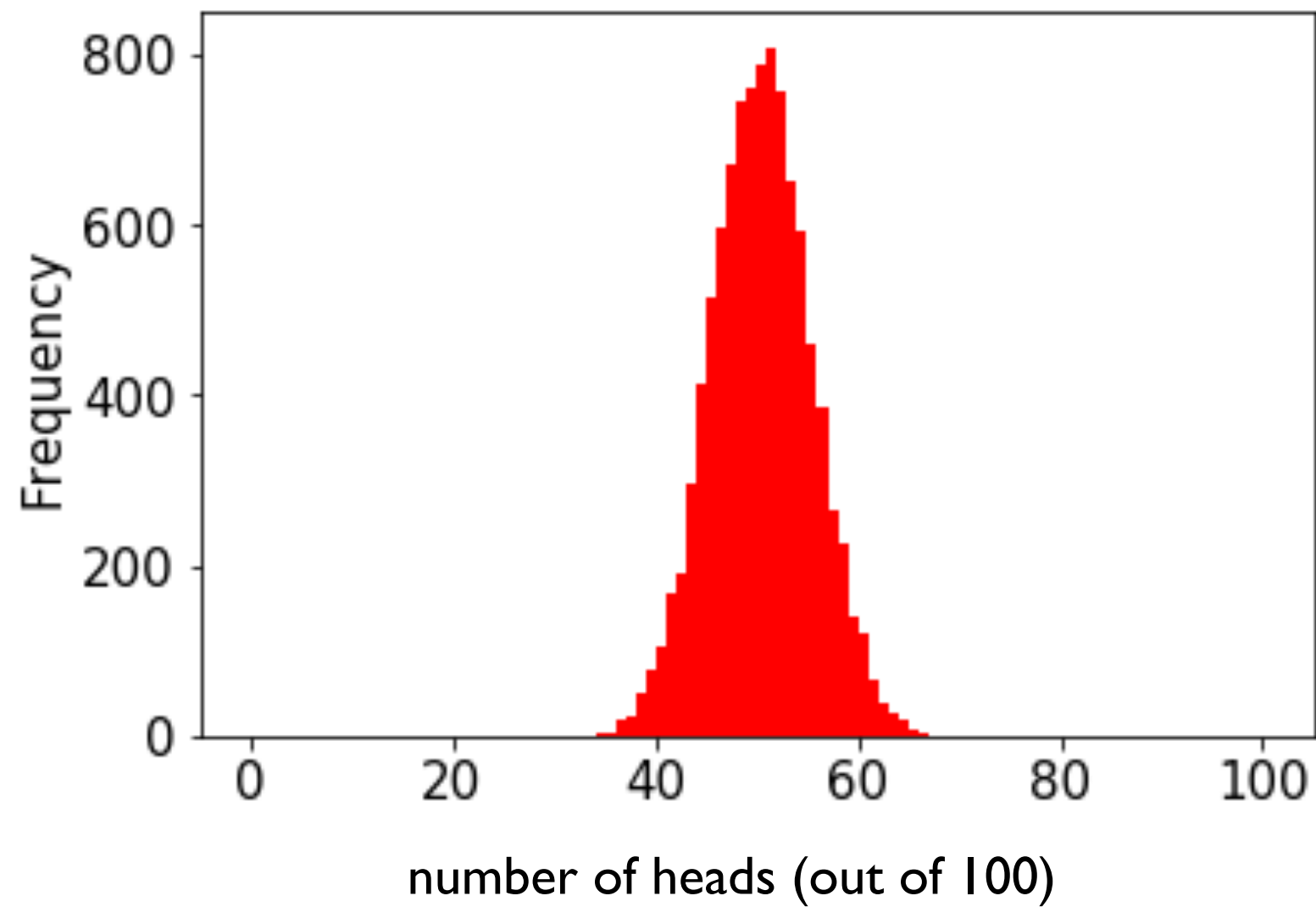
```
s = Series([0.1, 0, 1, 8, 9, 9.2])
```

```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=range(11))
```

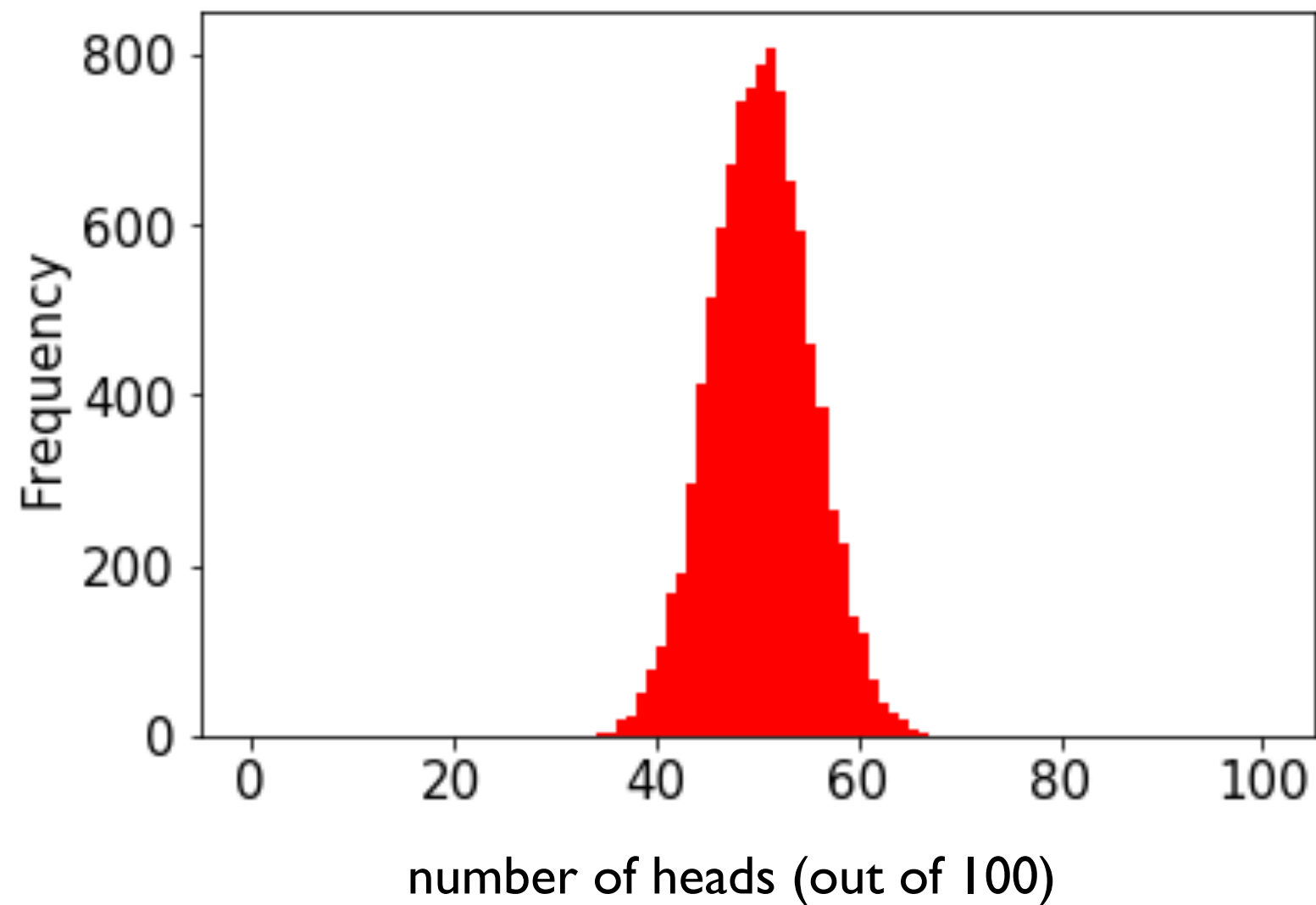


**this is easily done with range**

# Demo: Visualize CoinSim Results

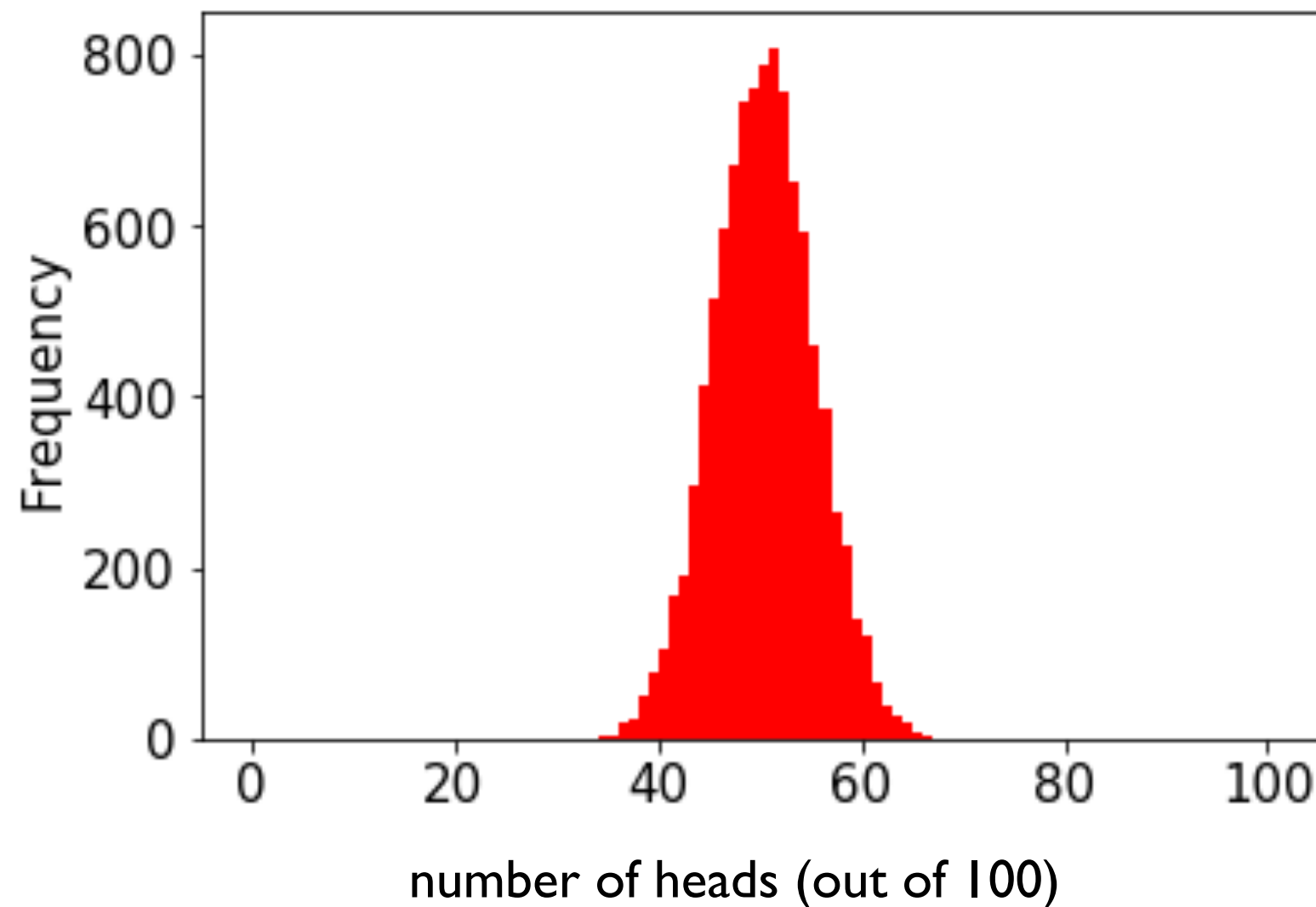


# Demo: Visualize CoinSim Results



this shape resembles what we often call a  
normal distribution or a "bell curve"

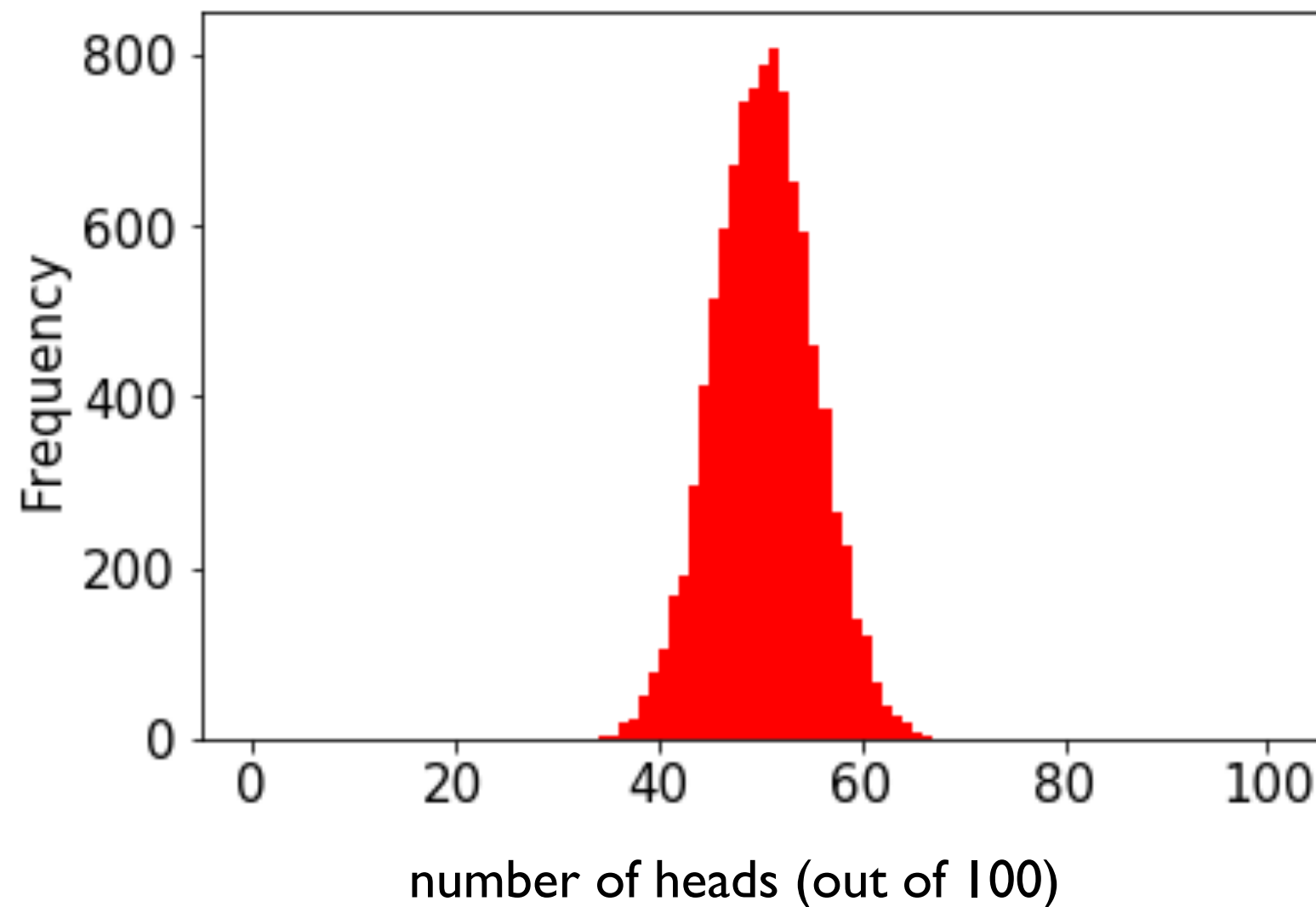
# Demo: Visualize CoinSim Results



this shape resembles what we often call a normal distribution or a "bell curve"

in general, if we take large samples enough times, the sample averages will look like this (we won't discuss exceptions here)

# Demo: Visualize CoinSim Results



numpy can directly  
generate random  
numbers fitting a  
normal distribution

this shape resembles what we often call a  
normal distribution or a "bell curve"

in general, if we take large samples enough  
times, the sample averages will look like this  
(we won't discuss exceptions here)

# Outline

choice()

bugs and seeding

significance

histograms

normal()

# normal

```
from numpy.random import choice, normal
import numpy as np

for i in range(10):
    print(normal())
```

# normal

```
from numpy.random import choice, normal
import numpy as np
```

```
for i in range(10):
    print(normal())
```

average is 0 (over many calls)

numbers closer to 0 more likely

-x just as likely as x

## Output:

-0.18638553993371157

0.02888452916769247

1.2474561113726423

-0.5388224399358179

-0.45143322136388525

-1.4001861112018241

0.28119371511868047

0.2608861898556597

-0.19246288728955144

0.2979572961710292



# normal

```
from numpy.random import choice, normal  
import numpy as np
```

```
s = Series(normal(size=10000))
```

# normal

```
from numpy.random import choice, normal  
import numpy as np
```

```
s = Series(normal(size=10000))
```

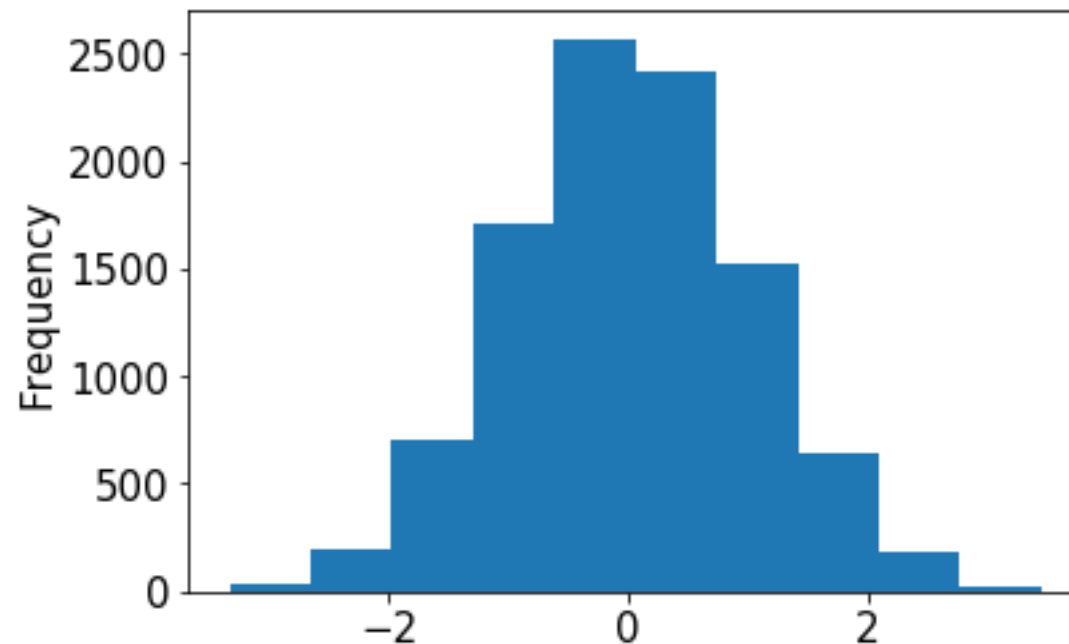
```
s.plot.hist()
```

# normal

```
from numpy.random import choice, normal  
import numpy as np
```

```
s = Series(normal(size=10000))
```

```
s.plot.hist()
```

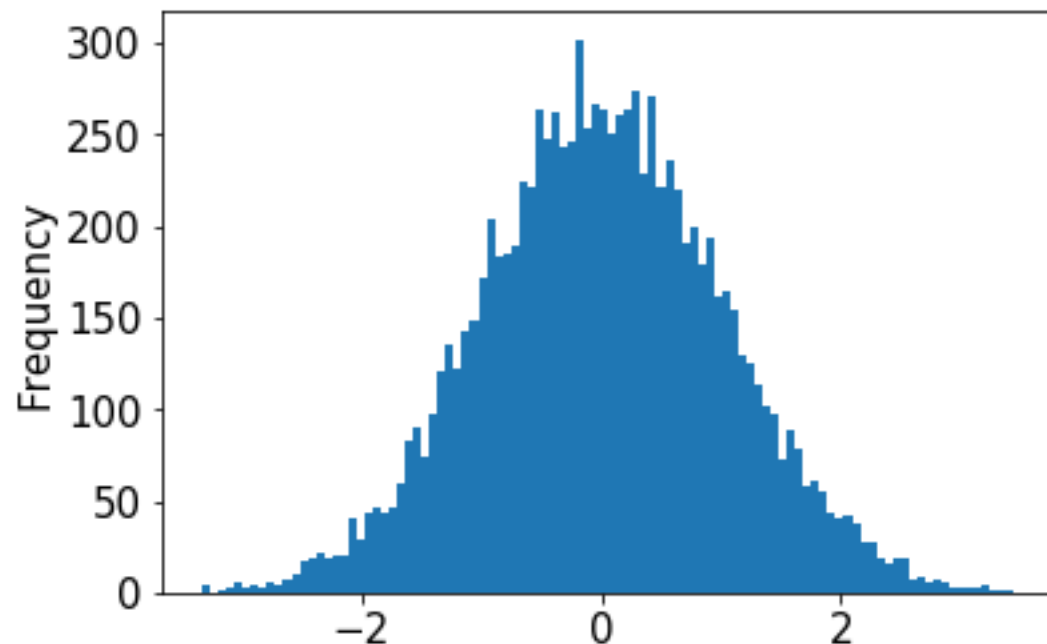


# normal

```
from numpy.random import choice, normal  
import numpy as np
```

```
s = Series(normal(size=10000))
```

```
s.plot.hist(bins=100)
```

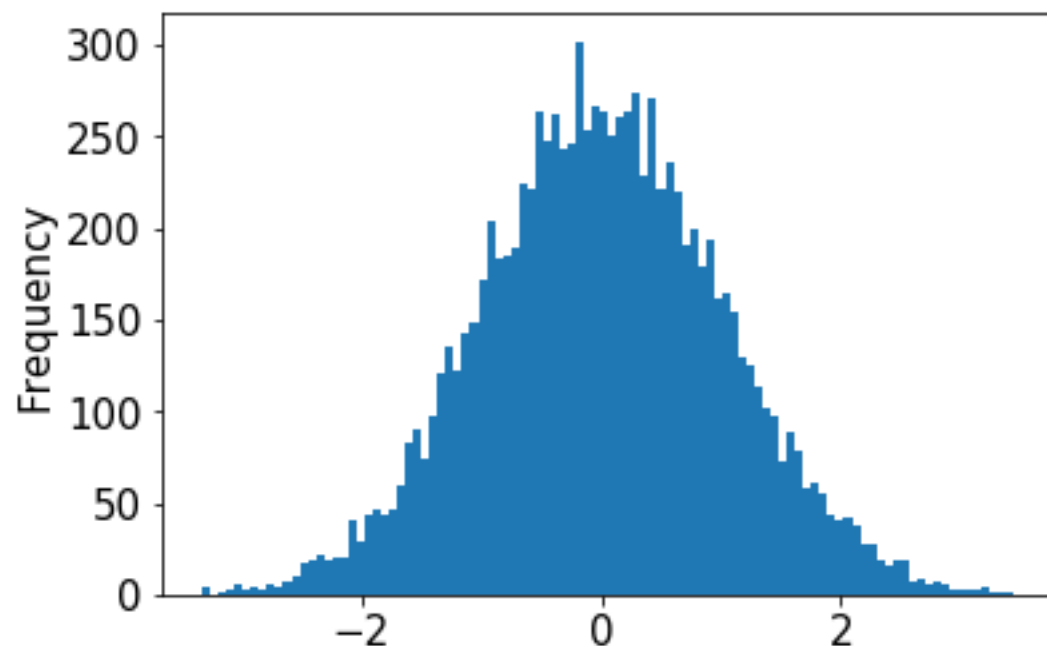


# normal

```
from numpy.random import choice, normal  
import numpy as np
```

```
s = Series(normal(size=10000))
```

```
s.plot.hist(bins=100, loc=, scale=)
```



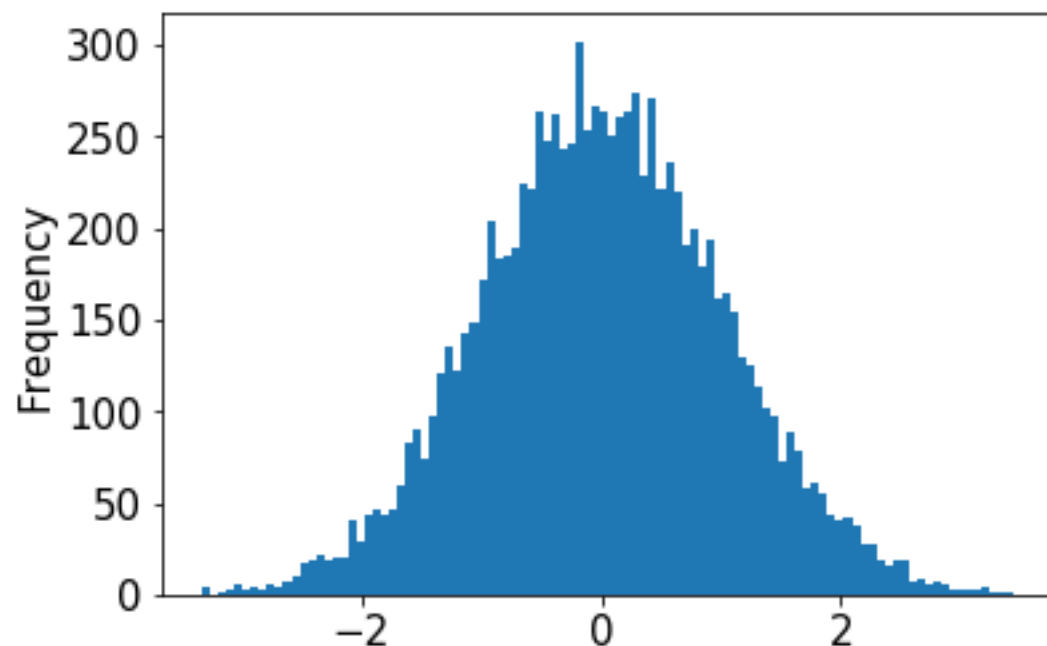
# normal

```
from numpy.random import choice, normal  
import numpy as np
```

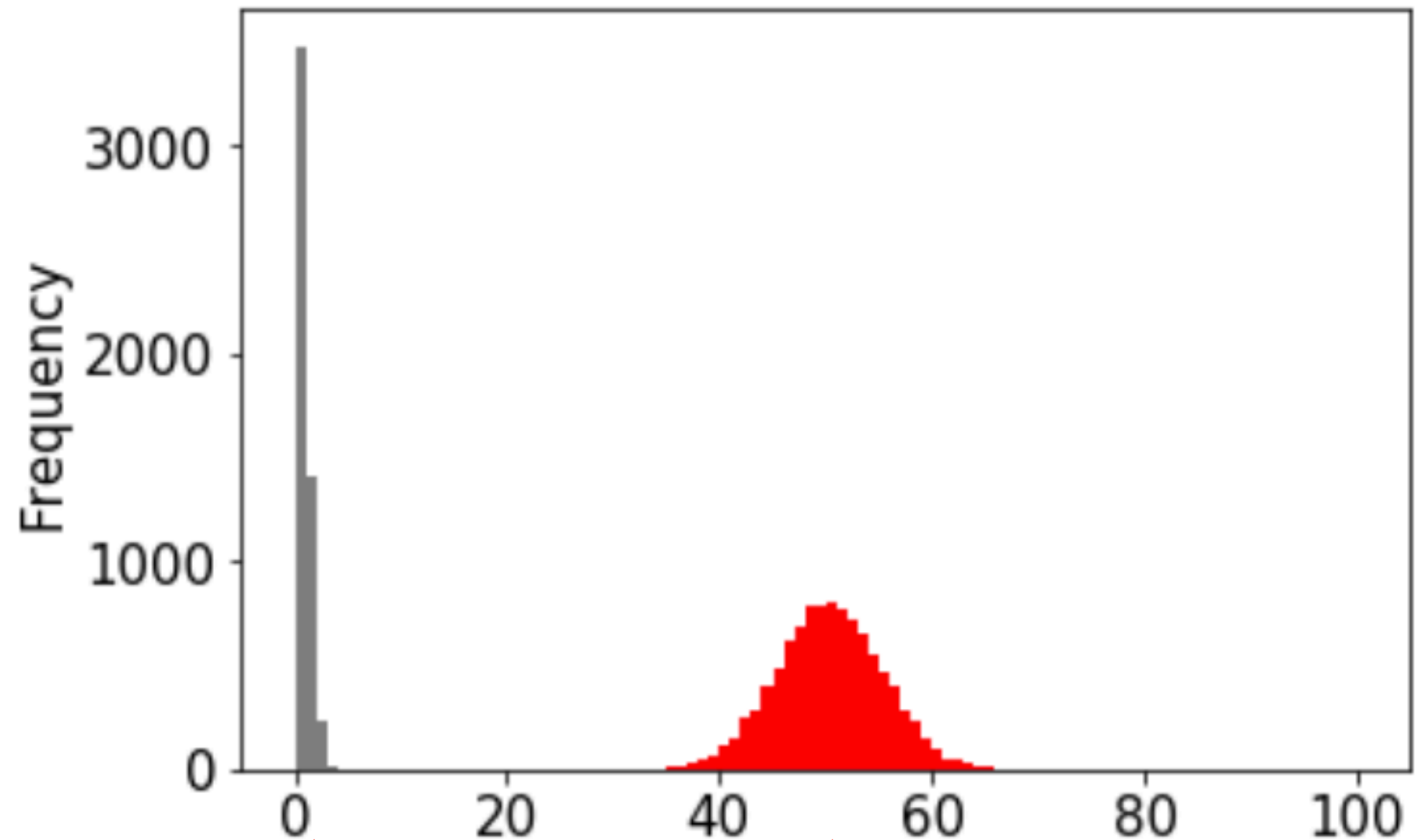
```
s = Series(normal(size=10000))
```

```
s.plot.hist(bins=100, loc=, scale=)
```

try plugging in different values  
(defaults are 0 and 1, respectively)



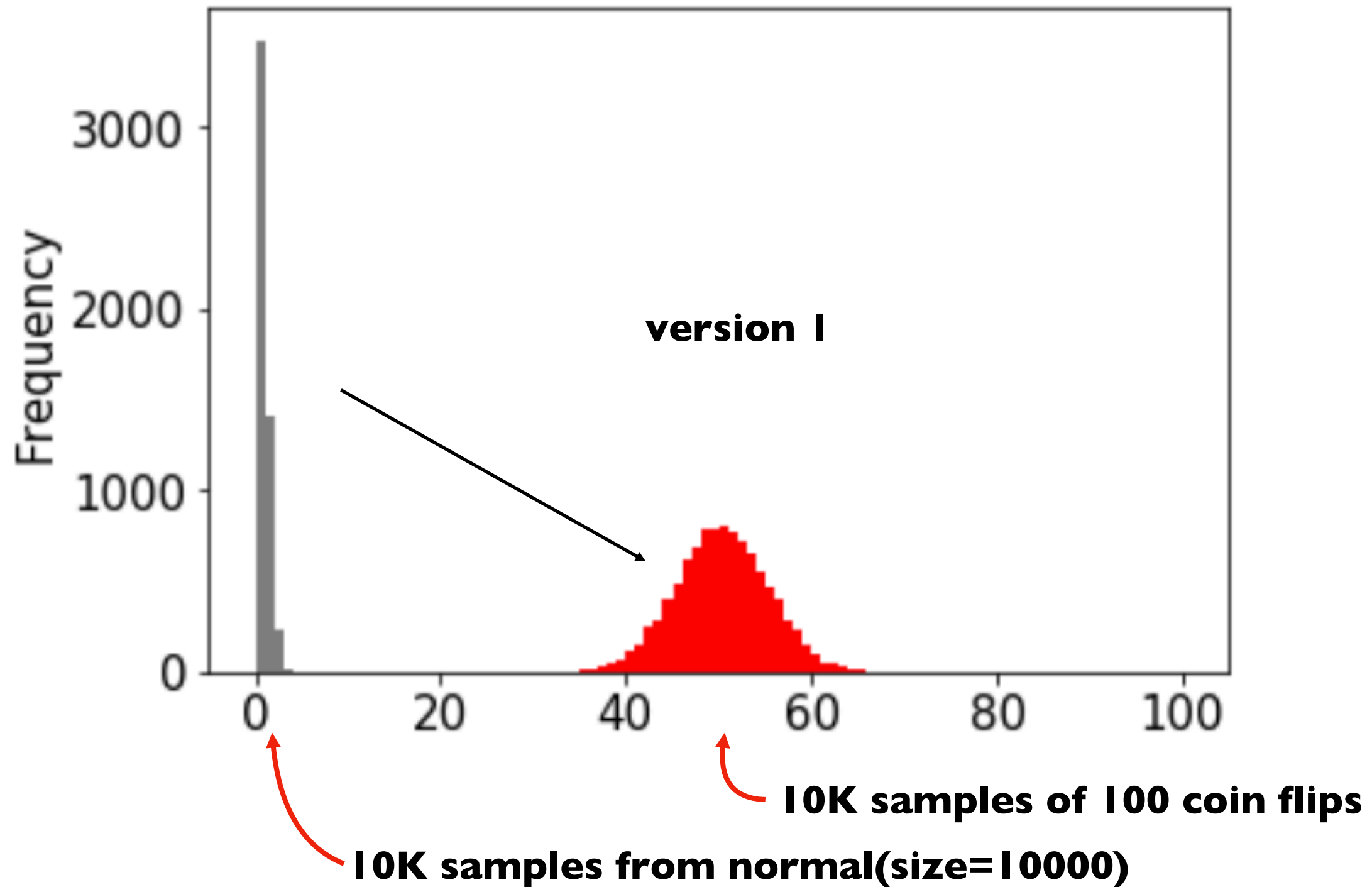
# Demo: plot overlay



**10K samples from normal(size=10000)**

**10K samples of 100 coin flips**

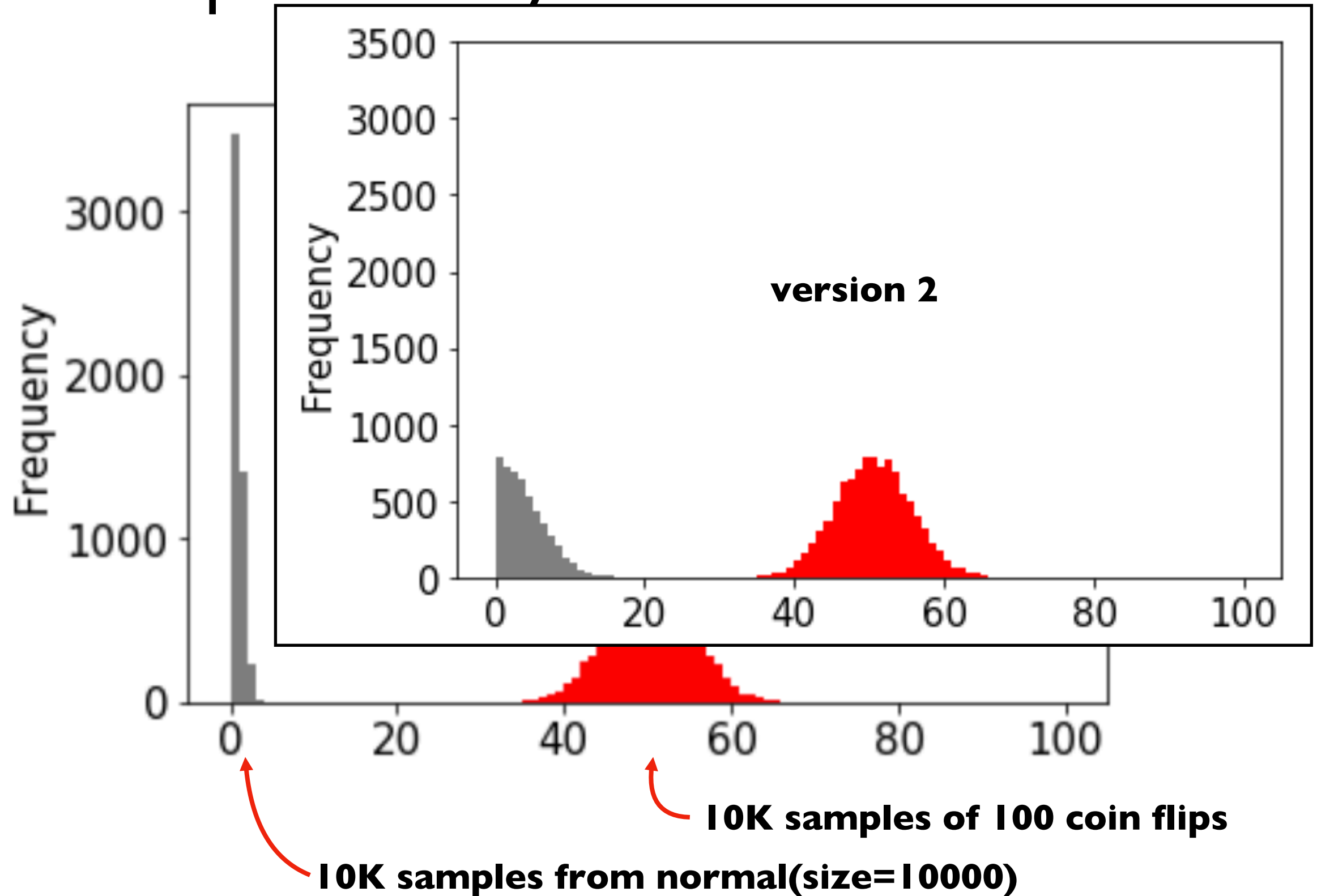
# Demo: plot overlay



**goal:** play with **loc** and **scale** arguments to normal until gray overlaps red

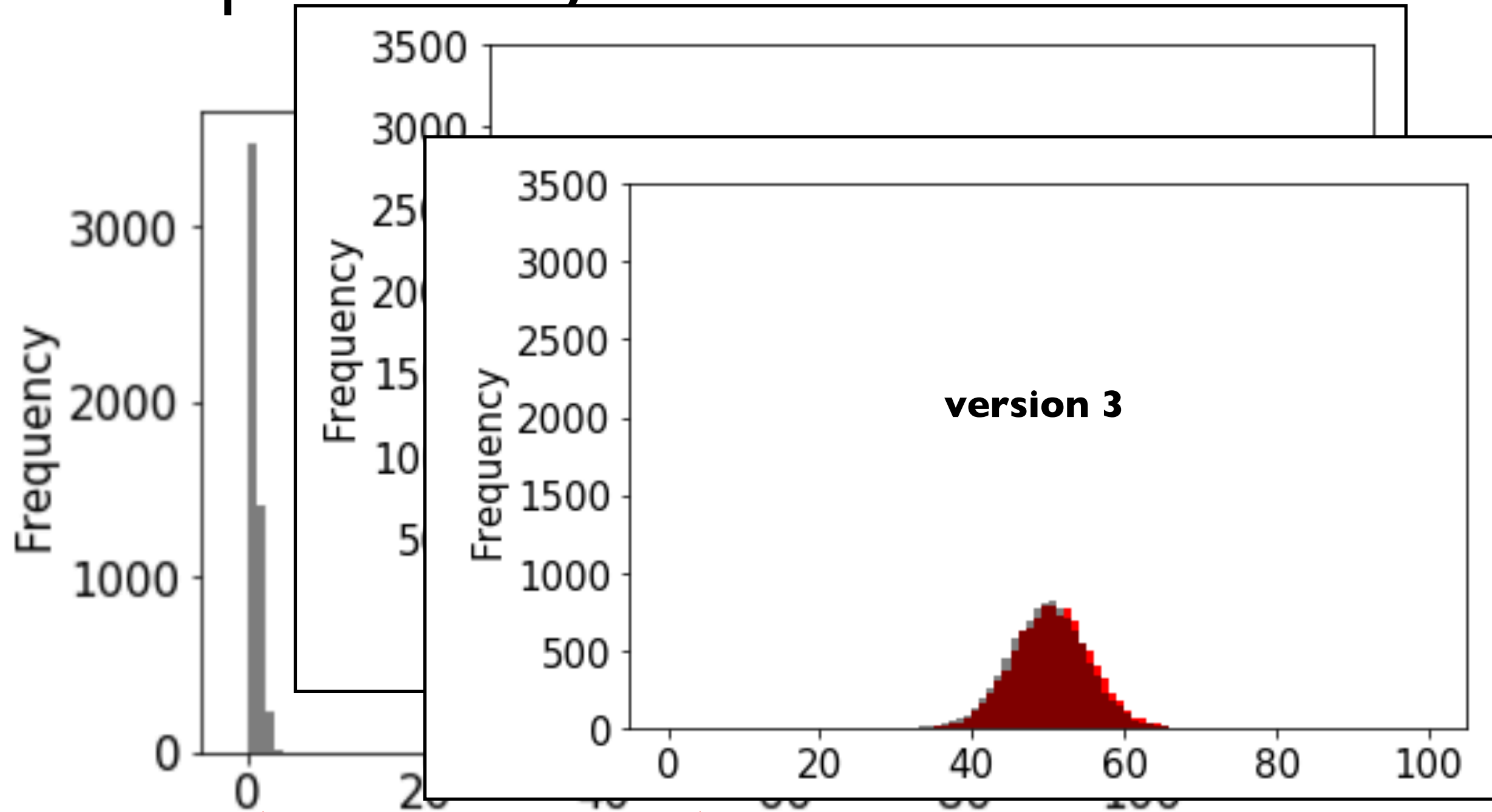


# Demo: plot overlay



**goal:** play with **loc** and **scale** arguments to normal until gray overlaps red

# Demo: plot overlay



**10K samples from normal(size=10000)**

**10K samples of 100 coin flips**

**goal:** play with **loc** and **scale** arguments to normal until gray overlaps red