

[301] Randomness

Tyler Caraza-Harter

Which series was randomly generated?

Which did I pick by hand?



TODO

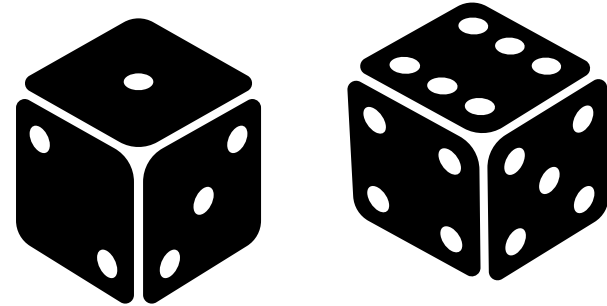
winter-break reading

wed review session: how to prep

Why Randomize?

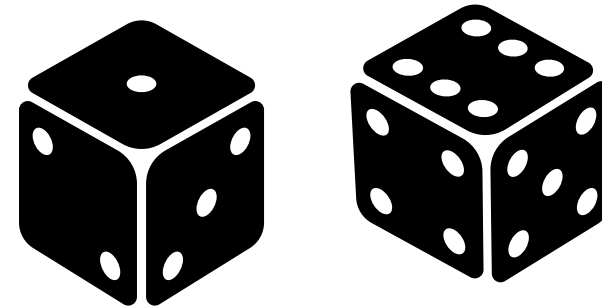
Why Randomize?

Games



Why Randomize?

Games

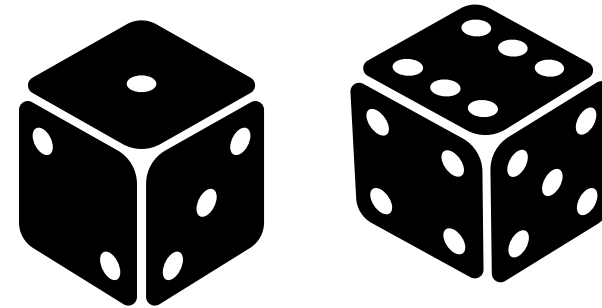


Security



Why Randomize?

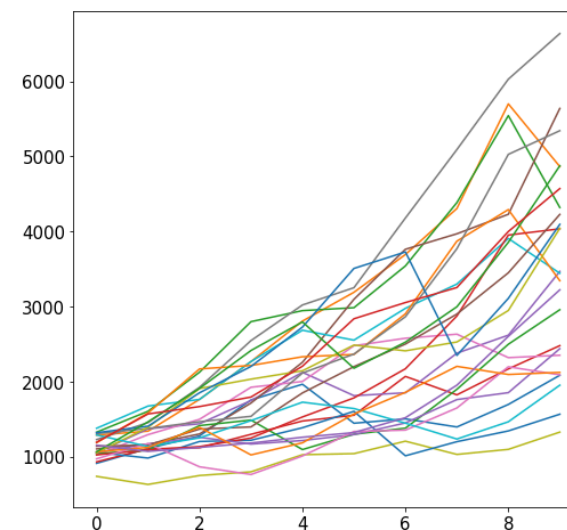
Games



Security

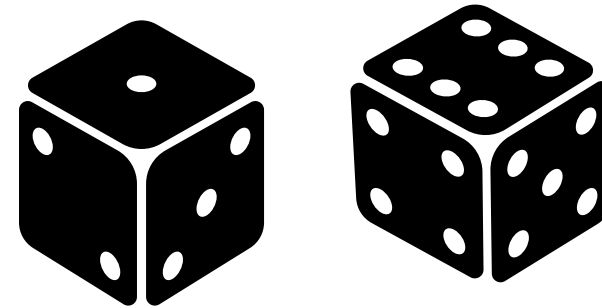


Simulation



Why Randomize?

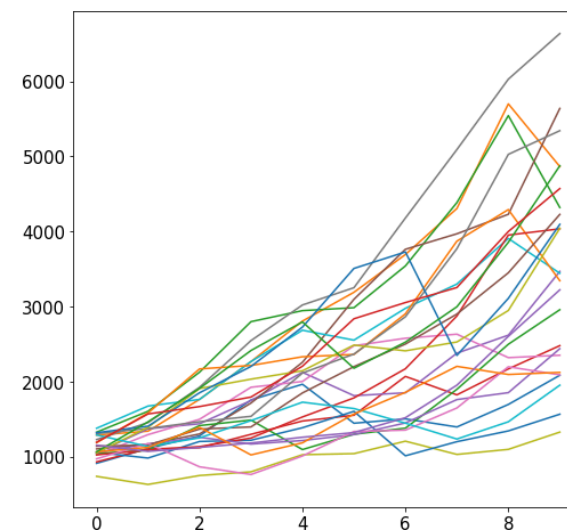
Games



Security



Simulation



our focus

Outline

choice()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

statistical significance: an intuitive approach

New Functions Today

Previous (from random module that comes w/ Python):

- **`choice`, `choices`, `randint`**

New Functions Today

Previous (from random module that comes w/ Python):

- **choice, choices, randint**

numpy.random:

- powerful collection of functions
- today: **choice, normal**

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

New Functions Today

Previous (from random module that comes w/ Python):

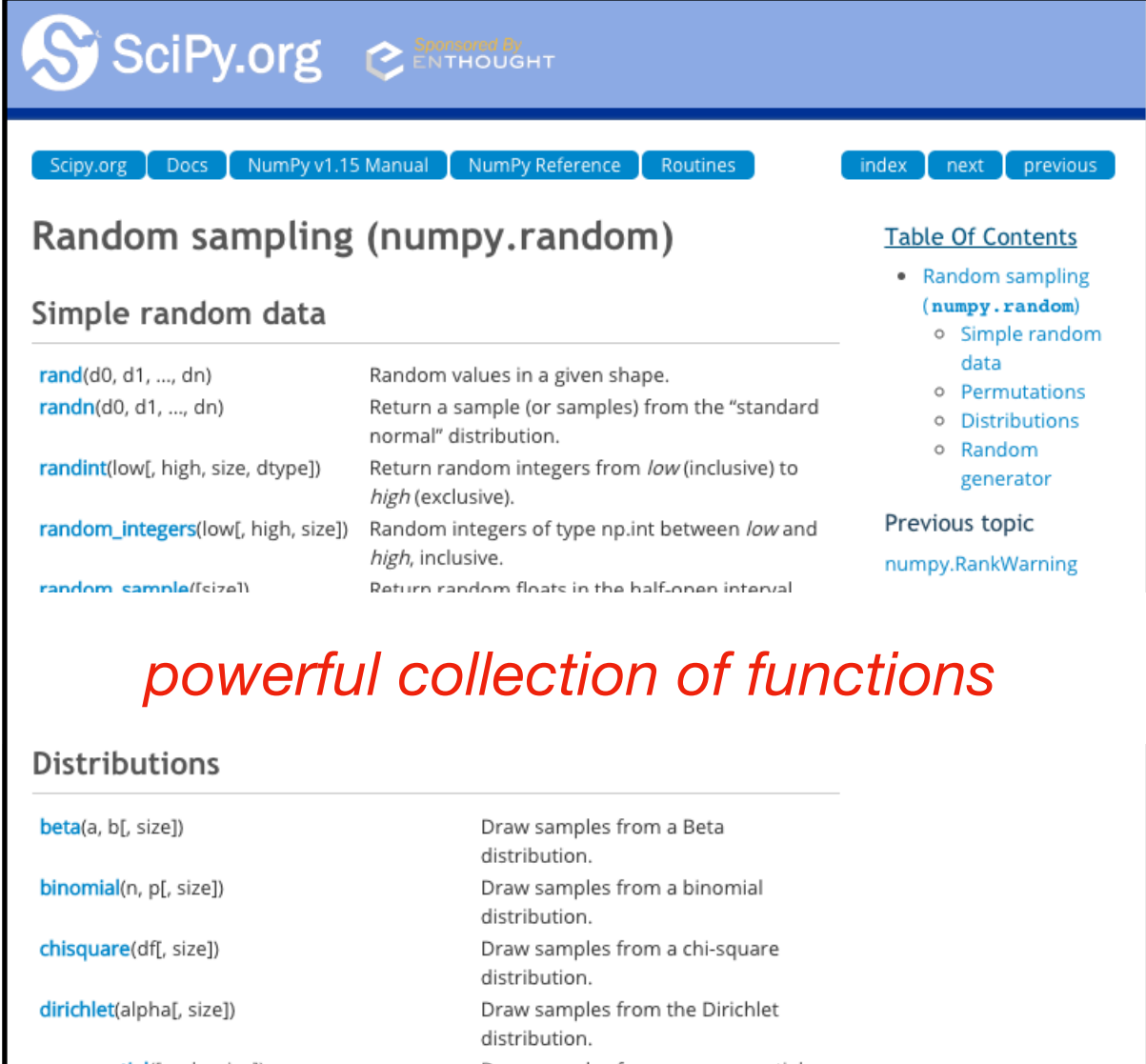
- **choice, choices, randint**

numpy.random:

- powerful collection of functions
- today: **choice, normal**

Series.line.hist:

- similar to bar plot
- visualize spread of random results



SciPy.org Sponsored By ENTHOUGHT

SciPy.org Docs NumPy v1.15 Manual NumPy Reference Routines index next previous

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 |

Table Of Contents

- Random sampling (`numpy.random`)
 - Simple random data
 - Permutations
 - Distributions
 - Random generator

Previous topic
[numpy.RankWarning](#)

powerful collection of functions

New Functions Today

Previous (from random module that comes w/ Python):

- **choice, choices, randint**

numpy.random:

- powerful collection of functions
- today: **choice** **normal**

Series.line.hist:

- similar to bar plot
- visualize spread of random results

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 |

powerful collection of functions

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 |

choice

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

choice

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

```
result = choice(          )
```

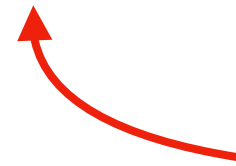


**list of things to
randomly choose from**

choice

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

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



**list of things to
randomly choose from**

choice

```
from numpy.random import choice, normal  
  
result = choice(["rock", "paper", "scissors"])  
print(result)
```

Output:

scissors

choice

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

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

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

Output:

```
scissors  
rock
```

choice

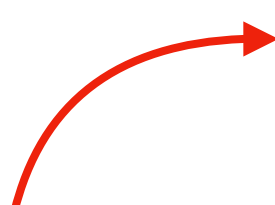
```
from numpy.random import choice, normal
```

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

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

choice

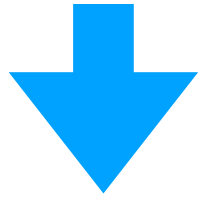
```
from numpy.random import choice, normal  
  
choice(["rock", "paper", "scissors"], size=5)
```

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

choice

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

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

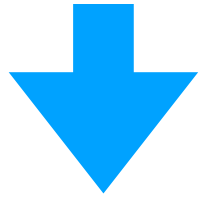


```
array(['rock', 'scissors', 'paper', 'rock', 'paper'], dtype='<U8')
```

choice

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

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



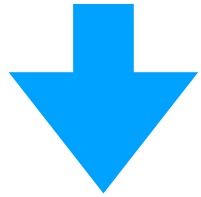
```
array(['rock', 'scissors', 'paper', 'rock', 'paper'], dtype='<U8')
```

1-dimensional ndarray with 5 items

choice

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

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



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

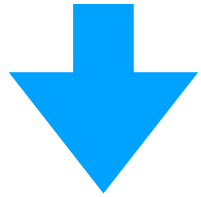
1-dimensional ndarray with 5 items

```
choice(["rock", "paper", "scissors"], size=(3,2))
```


choice

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

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

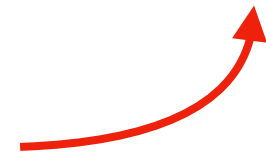


```
array(['rock', 'scissors', 'paper', 'rock', 'paper'], dtype='<U8')
```

1-dimensional ndarray with 5 items

```
choice(["rock", "paper", "scissors"], size=(3,2))
```

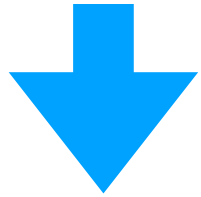
numpy shape tuple



choice

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

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



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

1-dimensional ndarray with 5 items

```
choice(["rock", "paper", "scissors"], size=(3,2))
```

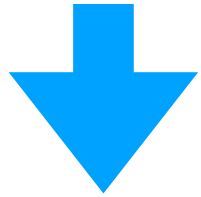


```
array([[ 'rock', 'scissors'],  
      [ 'paper', 'rock'],  
      [ 'scissors', 'paper']], dtype='<U8')
```

choice

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

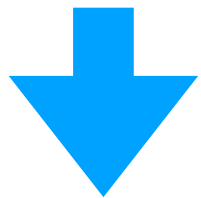
```
choice(["rock", "paper", "scissors"], size=5)
```



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

1-dimensional ndarray with 5 items

```
choice(["rock", "paper", "scissors"], size=(3,2))
```



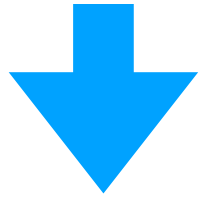
```
array([[ 'rock', 'scissors'],  
      [ 'paper', 'rock'],  
      [ 'scissors', 'paper']], dtype='<U8')
```

???-dimensional ndarray with ??? items

choice

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

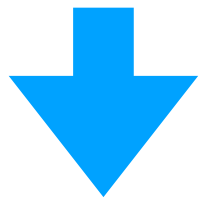
```
choice(["rock", "paper", "scissors"], size=5)
```



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

1-dimensional ndarray with 5 items

```
choice(["rock", "paper", "scissors"], size=(3,2))
```



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

2-dimensional ndarray with 6 items

Random values and Pandas

```
from numpy.random import choice, normal

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

Random values and Pandas

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

```
# random Series
```

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

Random values and Pandas

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

```
# random Series
```

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

| | |
|---------------|----------|
| 0 | paper |
| 1 | scissors |
| 2 | paper |
| 3 | rock |
| 4 | rock |
| dtype: object | |

Random values and Pandas

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

```
# random Series
```

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

| | |
|---------------|----------|
| 0 | paper |
| 1 | scissors |
| 2 | paper |
| 3 | rock |
| 4 | rock |
| dtype: object | |

```
# random DataFrame
```

```
DataFrame(choice(["rock", "paper", "scissors"], size=(5,3)))
```

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

Demo 1: exploring bias

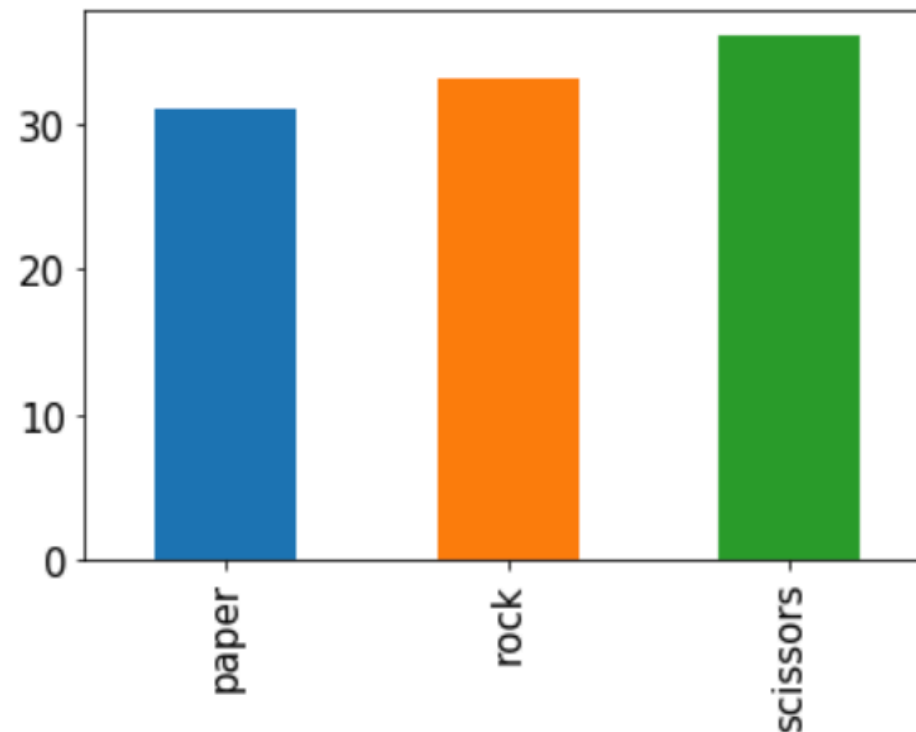
```
choice(["rock", "paper", "scissors"])
```

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

Demo 1: exploring bias

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

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

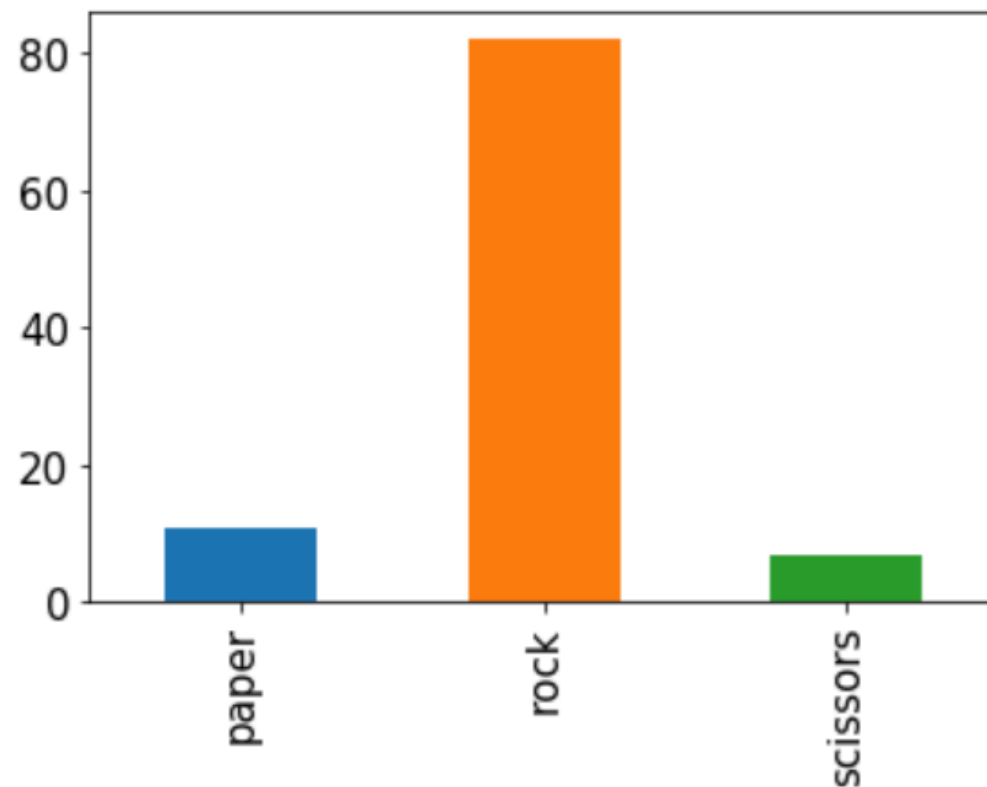


Demo 1: exploring bias

```
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)?



Random Strings vs. Random Ints

```
from numpy.random import choice, normal  
  
# random string: rock, paper, or scissors  
choice(["rock", "paper", "scissors"])
```

Random Strings vs. Random Ints

```
from numpy.random import choice, normal

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

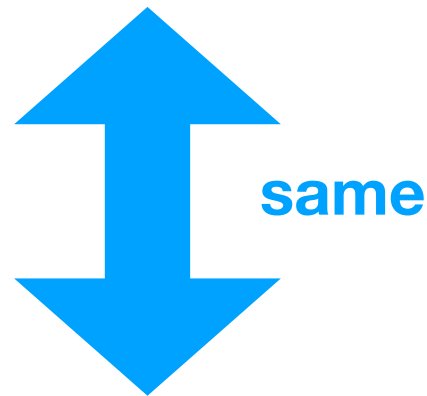
# random int: 0, 1, or 2
choice([0, 1, 2])
```

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])
```



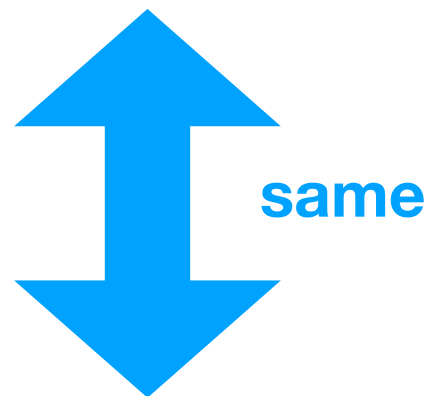
```
# random int (approach 2): 0, 1, or 2  
choice(3)
```

Random 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()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

statistical significance: an intuitive approach

Example: change over time

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

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

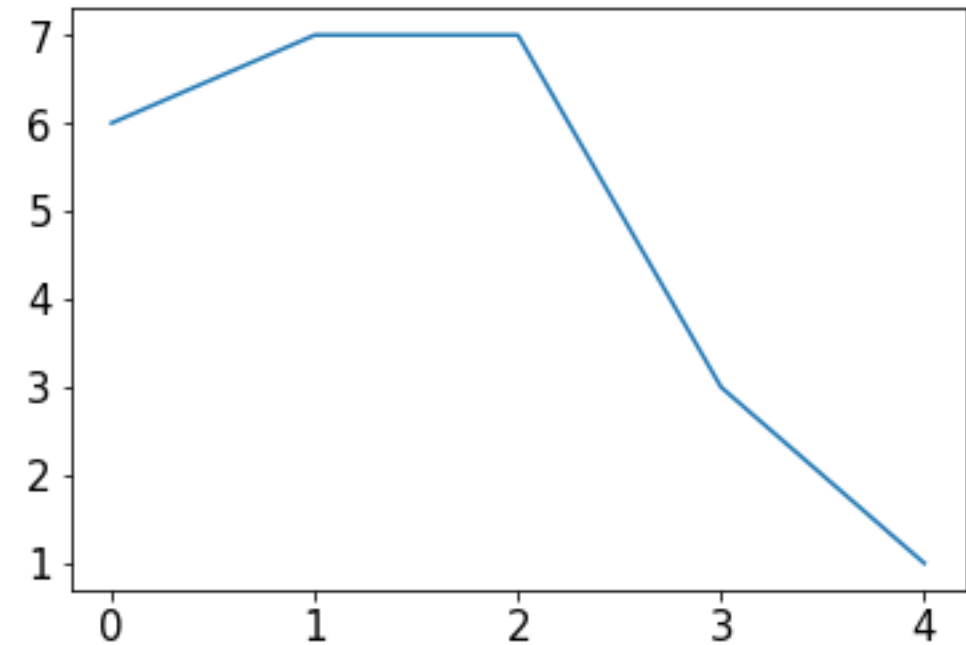
Example: change over time

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

| | |
|---|---|
| 0 | 6 |
| 1 | 7 |
| 2 | 7 |
| 3 | 3 |
| 4 | 1 |

dtype: int64

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

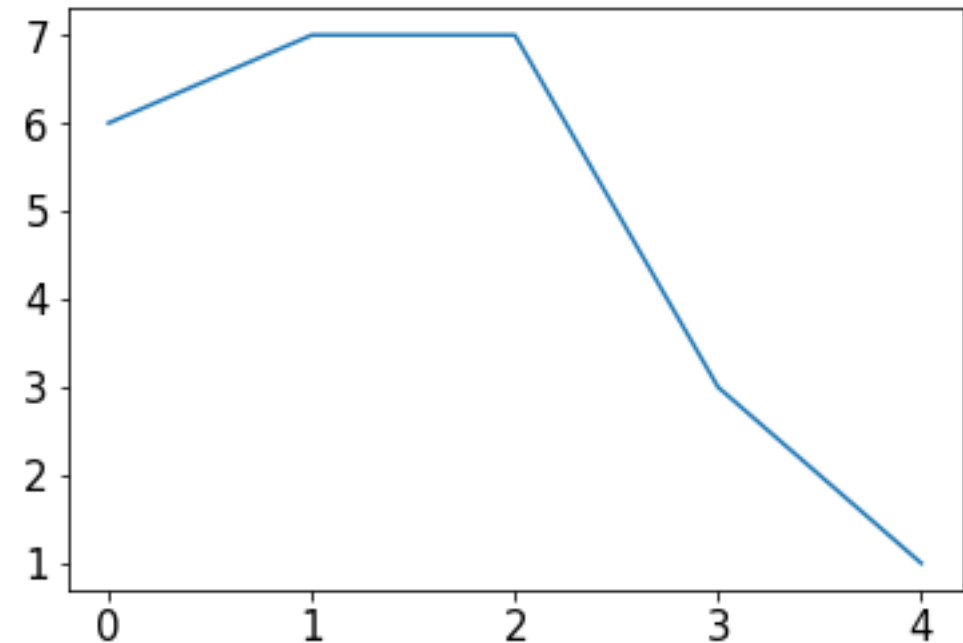


Example: change over time

```
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)
```

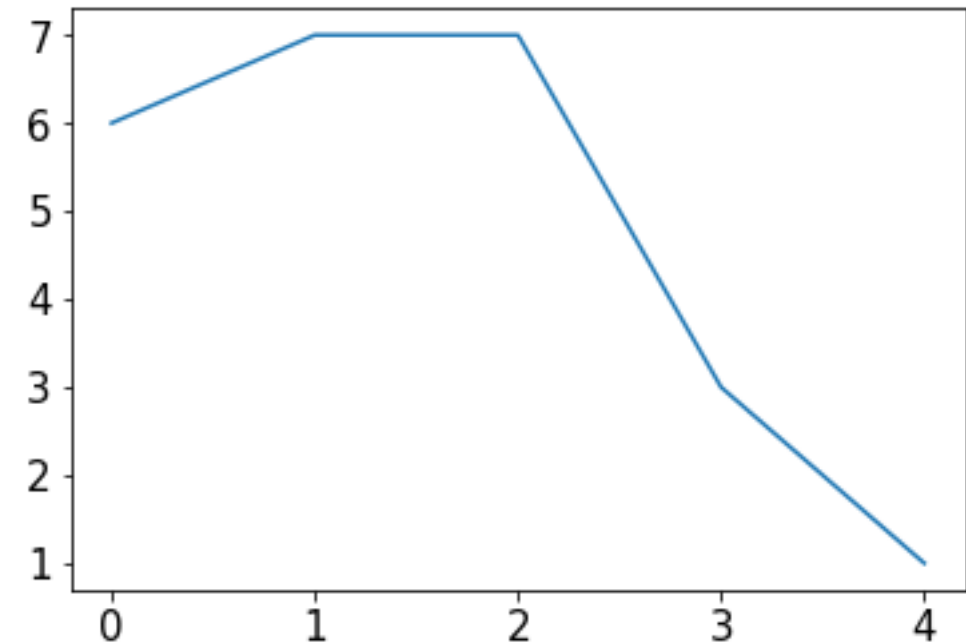
what are we computing for diff?

Example: change over time

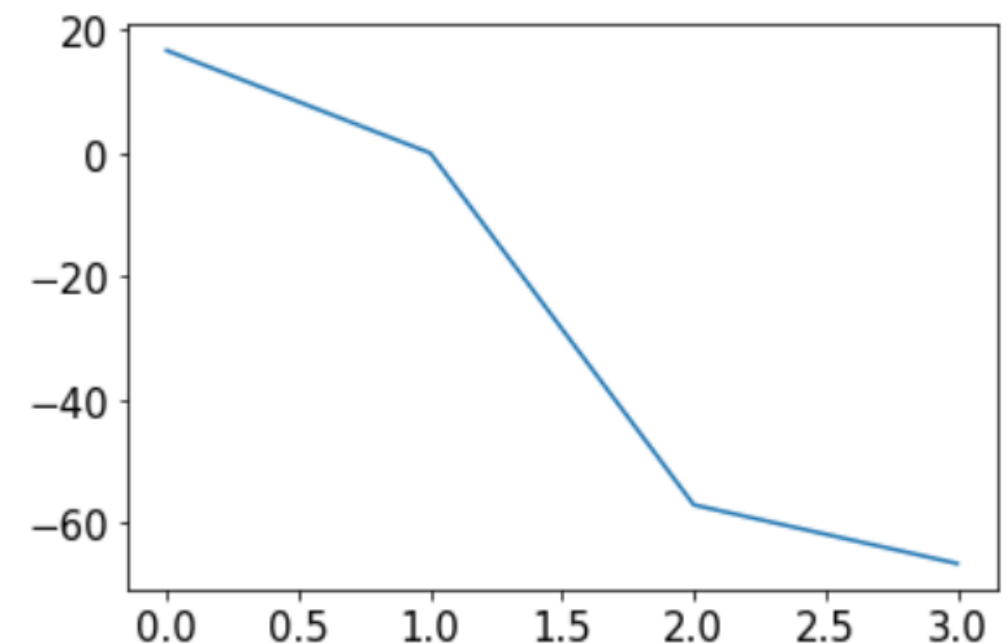
```
s = Series(choice(10, size=5))
```

| | |
|--------------|---|
| 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()
```

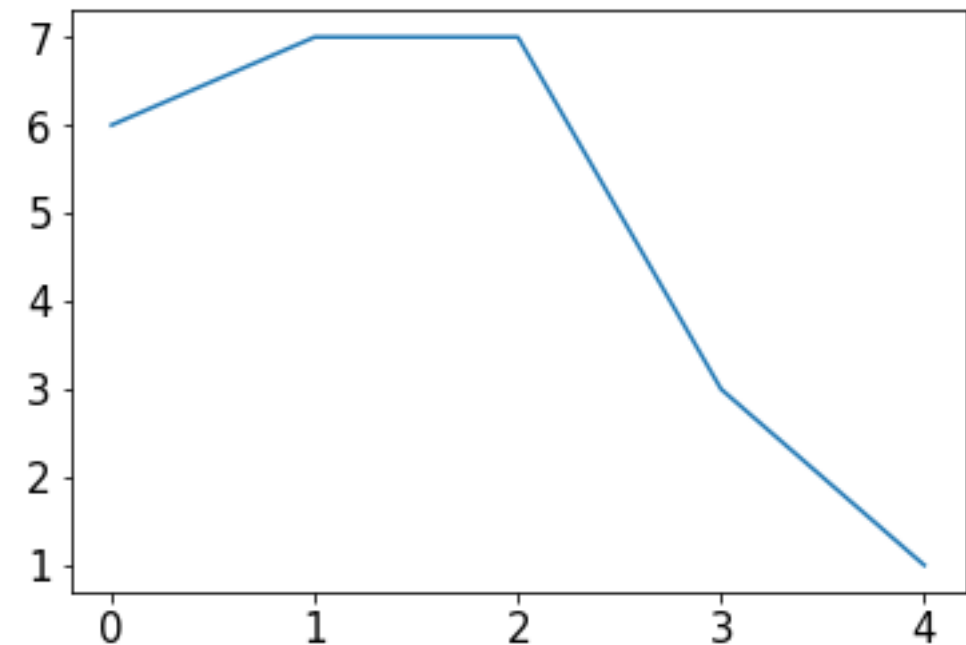


Example: change over time

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

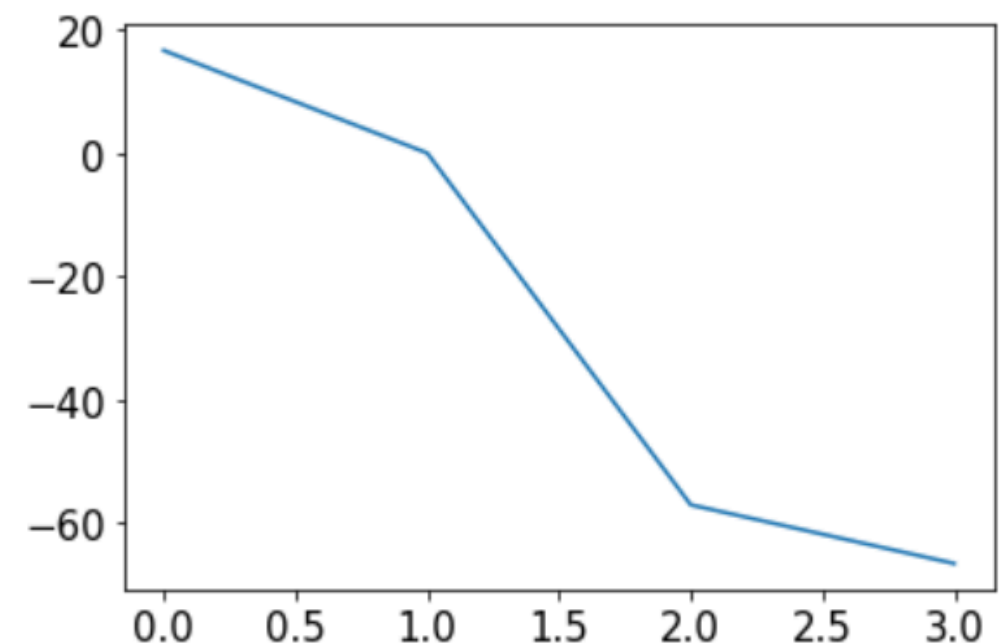
| | |
|--------------|---|
| 0 | 6 |
| 1 | 7 |
| 2 | 7 |
| 3 | 3 |
| 4 | 1 |
| dtype: int64 | |

```
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?

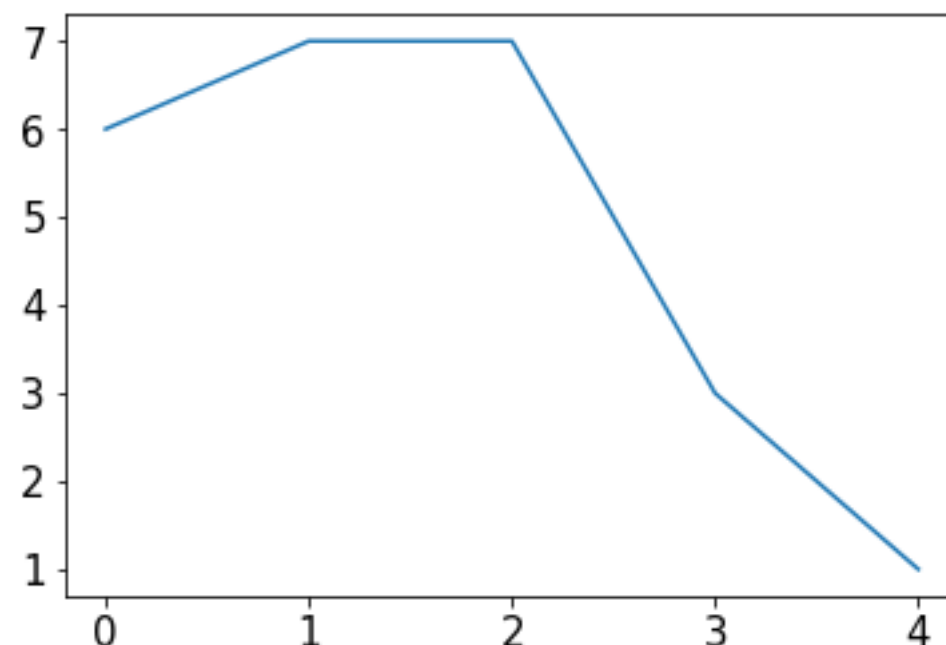


Example: change over time

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

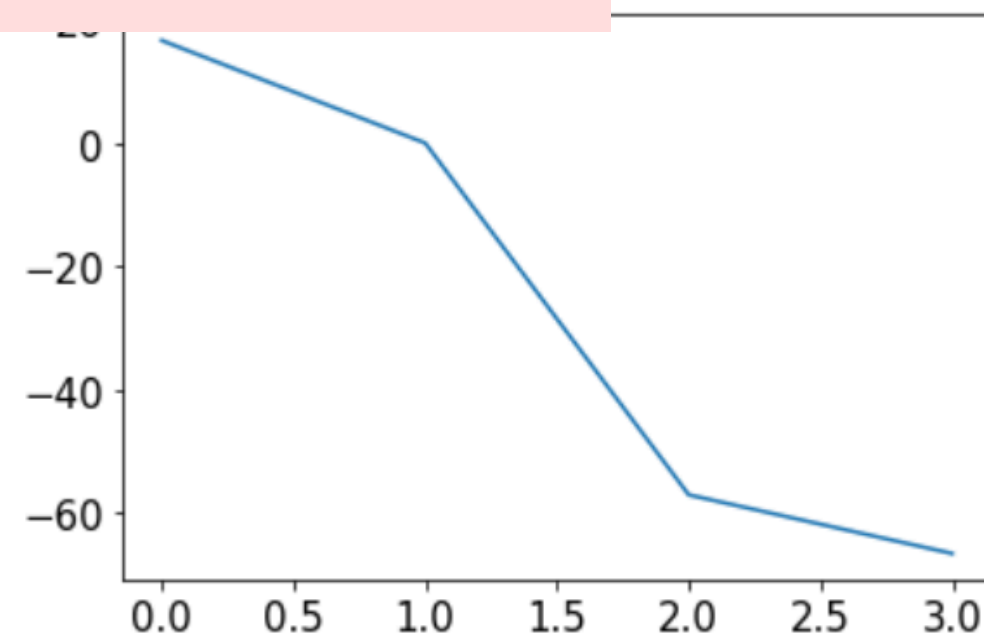
| | |
|--------------|---|
| 0 | 6 |
| 1 | 7 |
| 2 | 7 |
| 3 | 3 |
| 4 | 1 |
| dtype: int64 | |

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



```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:14:  
RuntimeWarning: divide by zero encountered in long_scalars
```

```
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?

Reproducibility

some bugs are easier to debug than others

- syntax or runtime errors easier than **semantic bugs**
- small inputs are easier than **big inputs**

a bug is **reproducible** if it shows up every time you run the program with the same inputs

Reproducibility

some bugs are easier to debug than others

- syntax or runtime errors easier than **semantic bugs**
- small inputs are easier than **big inputs**

a bug is **reproducible** if it shows up every time you run the program with the same inputs

who had a non-reproducible bug for a project this semester?

Reproducibility

some bugs are easier to debug than others

- syntax or runtime errors easier than **semantic bugs**
- small inputs are easier than **big inputs**

a bug is **reproducible** if it shows up every time you run the program with the same inputs

who had a non-reproducible bug for a project this semester?

non-reproducible bugs

- are **hard to fix**
- **common** with programs based on randomness

Reproducibility

some bugs are easier to debug than others

- syntax or runtime errors easier than **semantic bugs**
- small inputs are easier than **big inputs**

a bug is **reproducible** if it shows up every time you run the program with the same inputs

who had a non-reproducible bug for a project this semester?

non-reproducible bugs

- are **hard to fix**
- **common** with programs based on randomness

fortunately, the random values we've been generating are not really, truly random. They're merely **pseudorandom**.

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions more** ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions** more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions** more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions** more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

restart!

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions more** ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions more** ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... **billions** more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... billions more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

684, 559, 629, 192, 835, ...

37, 235, 908, 72, 767, ...

168, 527, 493, 584, 534, ...

874, 664, 249, 643, 952, ...

122, 174, 439, 709, 897, ...

867, 206, 701, 998, 118, ...

906, 713, 227, 980, 618, ...

... billions more ...

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Pseudorandom

0: 684, 559, 629, 192, 835, ...

1: 37, 235, 908, 72, 767, ...

2: 168, 527, 493, 584, 534, ...

3: 874, 664, 249, 643, 952, ...

4: 122, 174, 439, 709, 897, ...

5: 867, 206, 701, 998, 118, ...

6: 906, 713, 227, 980, 618, ...

... billions more ...

seed

pseudorandom generators

- can generate **billions** of different **seemingly random sequences**
- subsequent calls to choice progress along these sequences
- every program run starts with a different sequence
- we can choose our sequence

Seeding

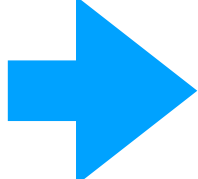
```
from numpy.random import choice, normal  
import numpy as np
```

```
np.random.seed(1)  
choice(10, size=5) ➡ array([5, 8, 9, 5, 0])
```

Seeding

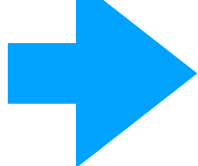
```
from numpy.random import choice, normal  
import numpy as np
```

```
np.random.seed(1)  
choice(10, size=5)
```



```
array([5, 8, 9, 5, 0])
```

```
np.random.seed(2)  
choice(10, size=5)
```

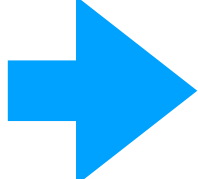


```
array([8, 8, 6, 2, 8])
```

Seeding

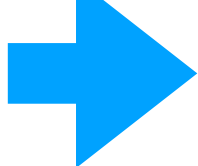
```
from numpy.random import choice, normal  
import numpy as np
```

```
np.random.seed(1)  
choice(10, size=5)
```



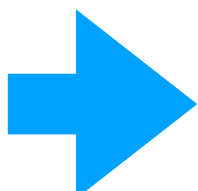
```
array([5, 8, 9, 5, 0])
```

```
np.random.seed(2)  
choice(10, size=5)
```



```
array([8, 8, 6, 2, 8])
```

```
np.random.seed(1)  
choice(10, size=5)
```

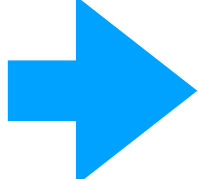


```
array([5, 8, 9, 5, 0])
```

Seeding

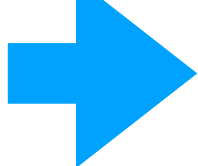
```
from numpy.random import choice, normal  
import numpy as np
```

```
np.random.seed(1)  
choice(10, size=5)
```



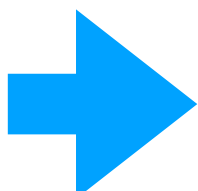
```
array([5, 8, 9, 5, 0])
```

```
np.random.seed(2)  
choice(10, size=5)
```



```
array([8, 8, 6, 2, 8])
```

```
np.random.seed(1)  
choice(10, size=5)
```



```
array([5, 8, 9, 5, 0])
```


Seeding

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

```
np.random.seed(1)  
choice(10, size=5) → array([5, 8, 9, 5, 0])
```

```
np.random.seed(2)  
choice(10, size=5) → array([8, 8, 6, 2, 8])
```

```
np.random.seed(1)  
choice(10, size=5) → array([5, 8, 9, 5, 0])
```

Debug tip: if you have a bug related to randomness, find a seed that causes the bug to arise, then use that seed until you find the problem.
(don't forget to remove it when you're done!)

Outline

choice()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

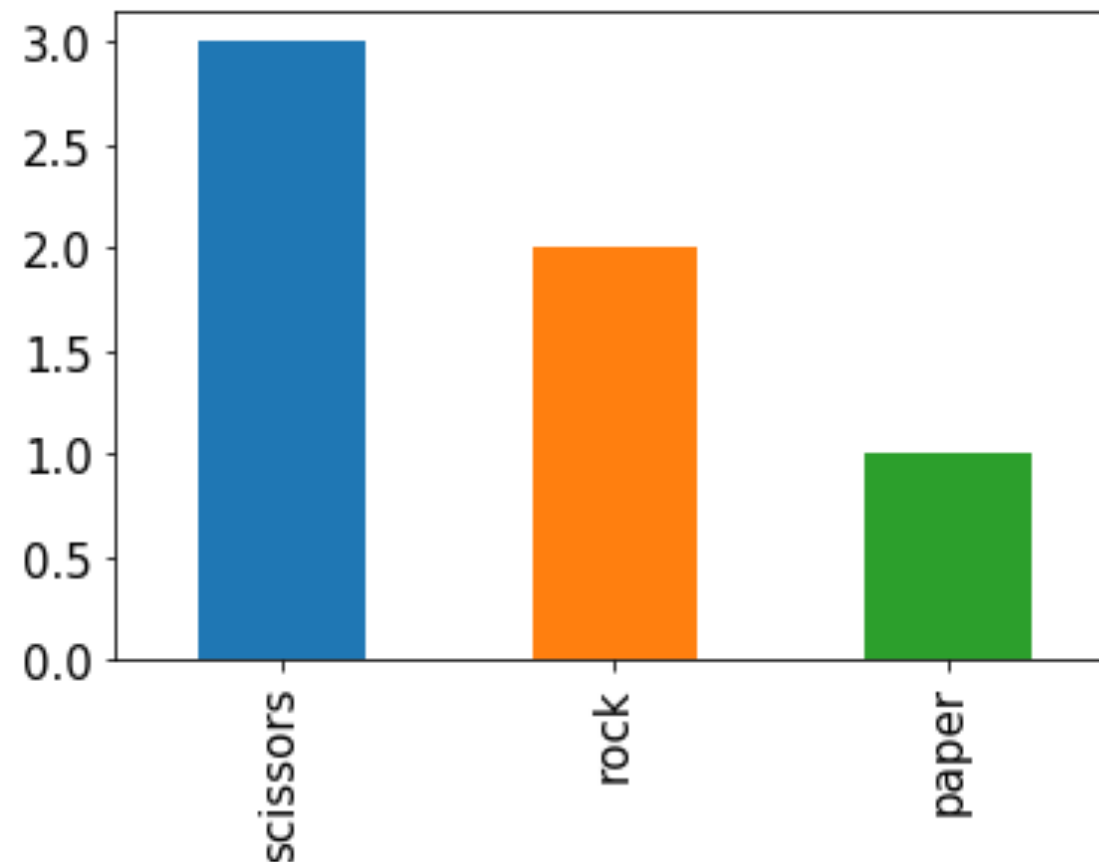
statistical significance: an intuitive approach

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()
```

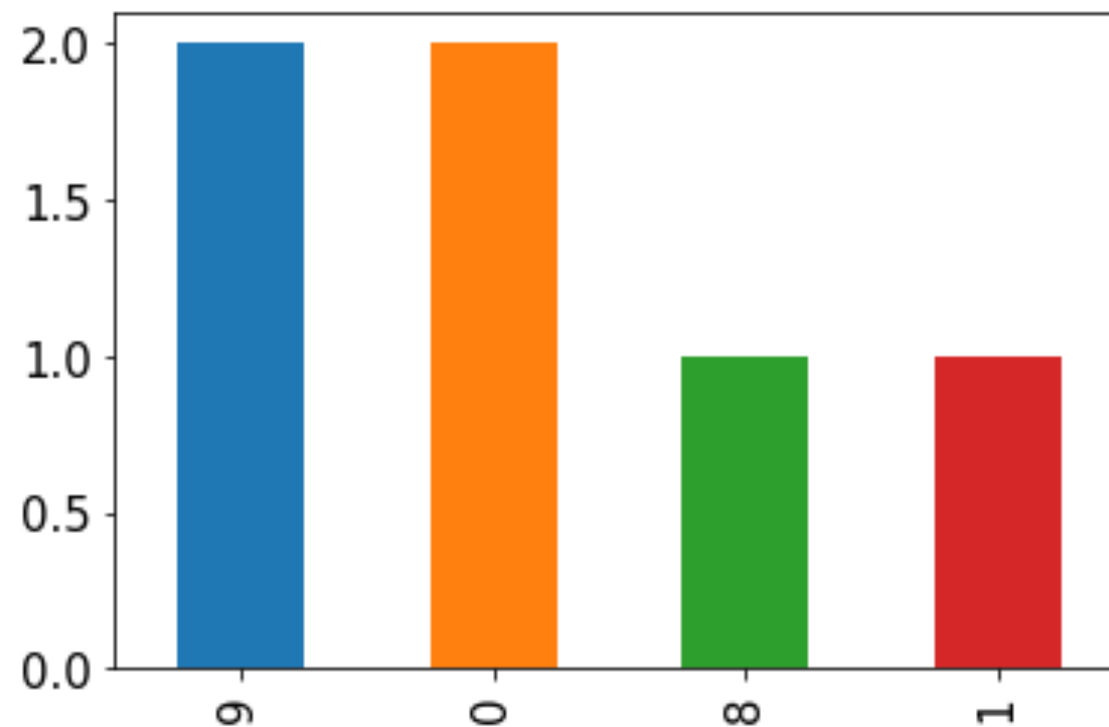


Frequencies across numbers

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

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

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



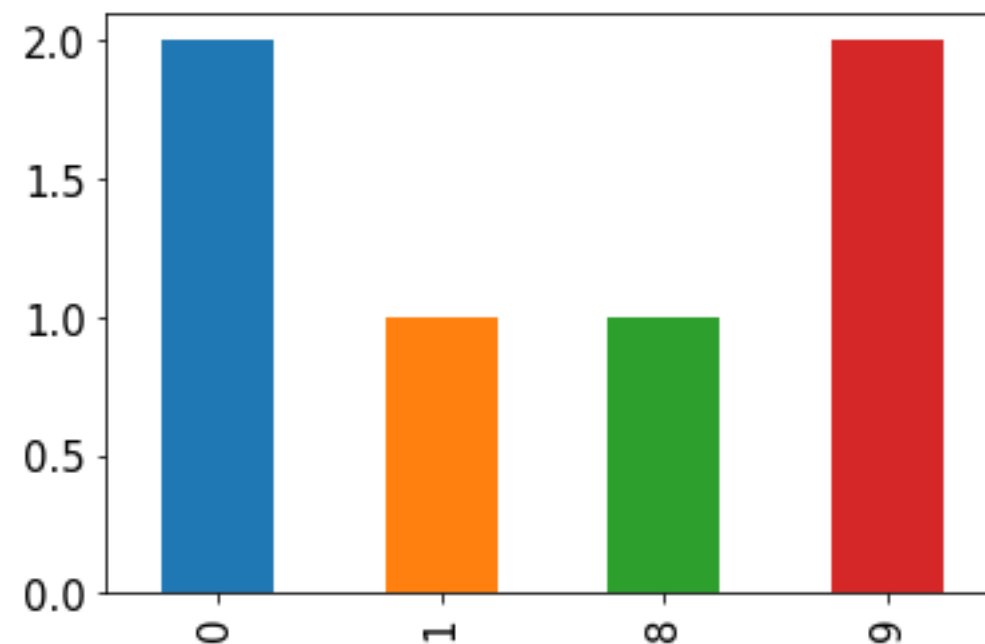
numbers not ordered

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()
```



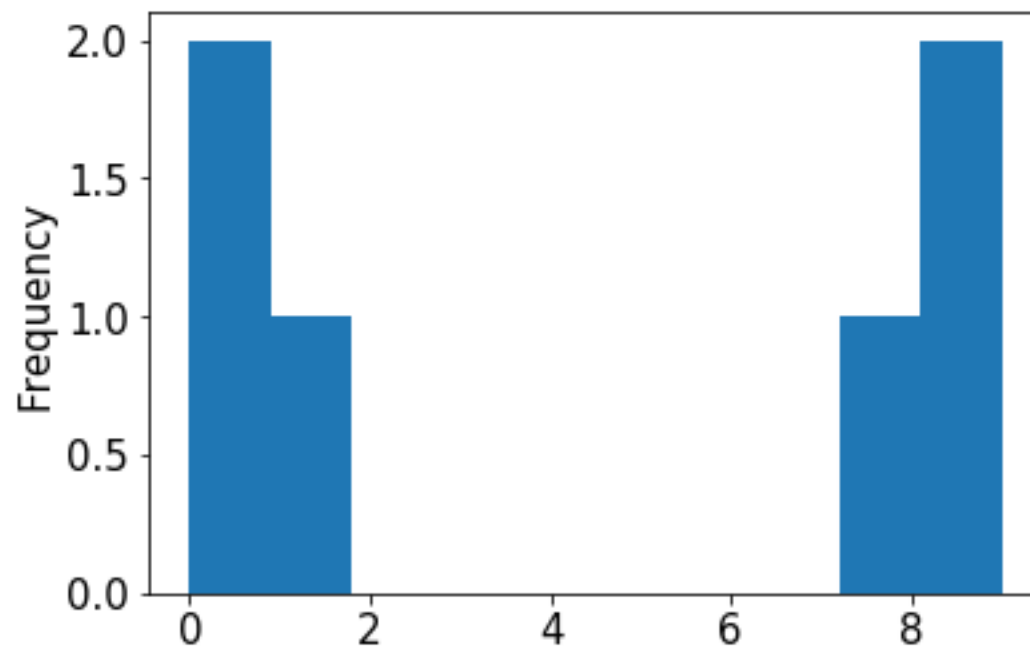
gap between 1 and 8 not obvious

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()
```

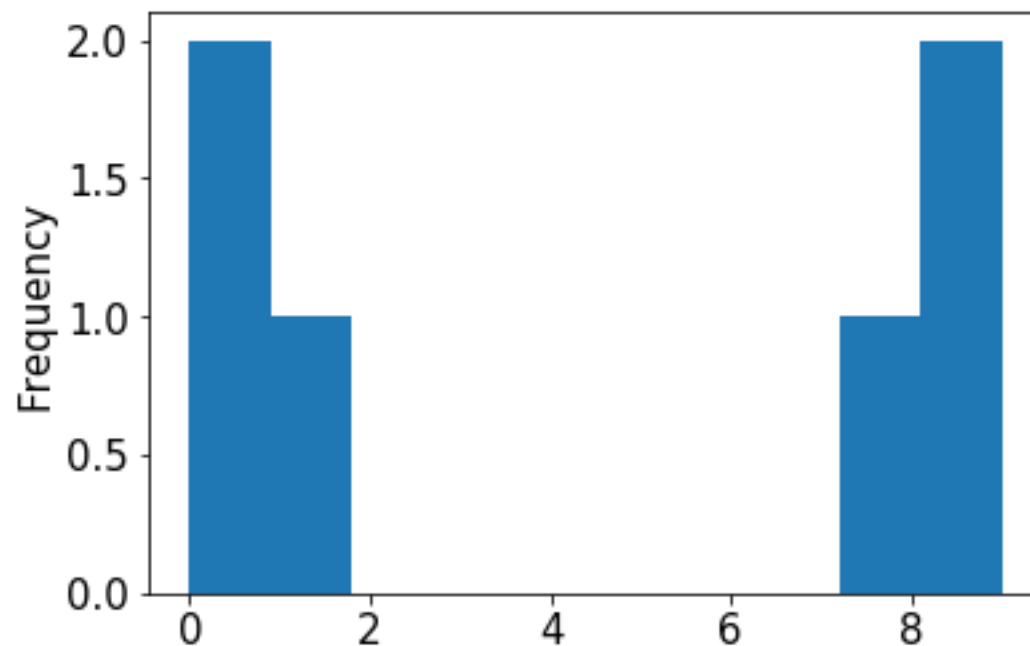


Frequencies across numbers

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()
```



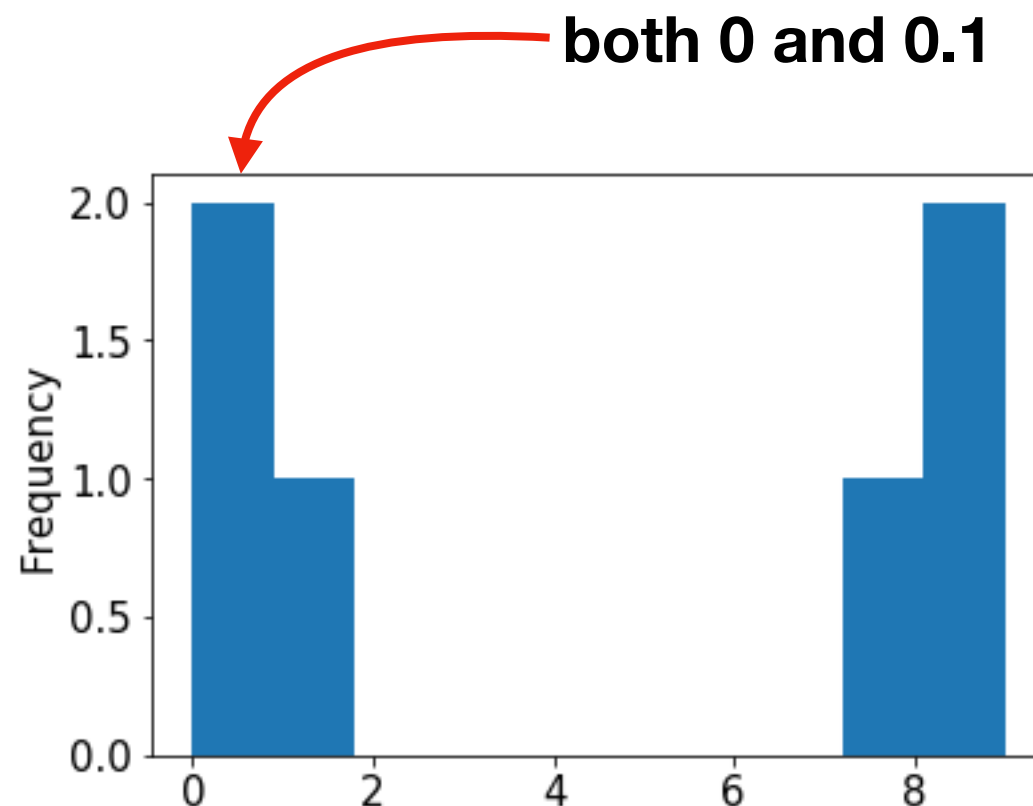
this kind of plot is called a histogram

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()
```



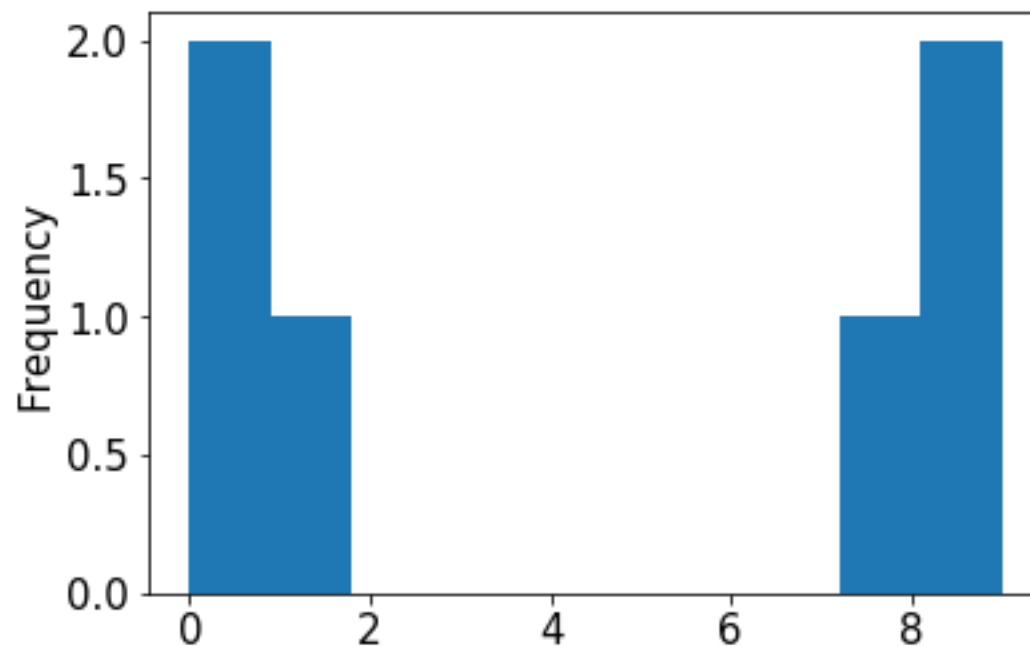
a histogram "bins" nearby numbers to create discrete bars

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)
```



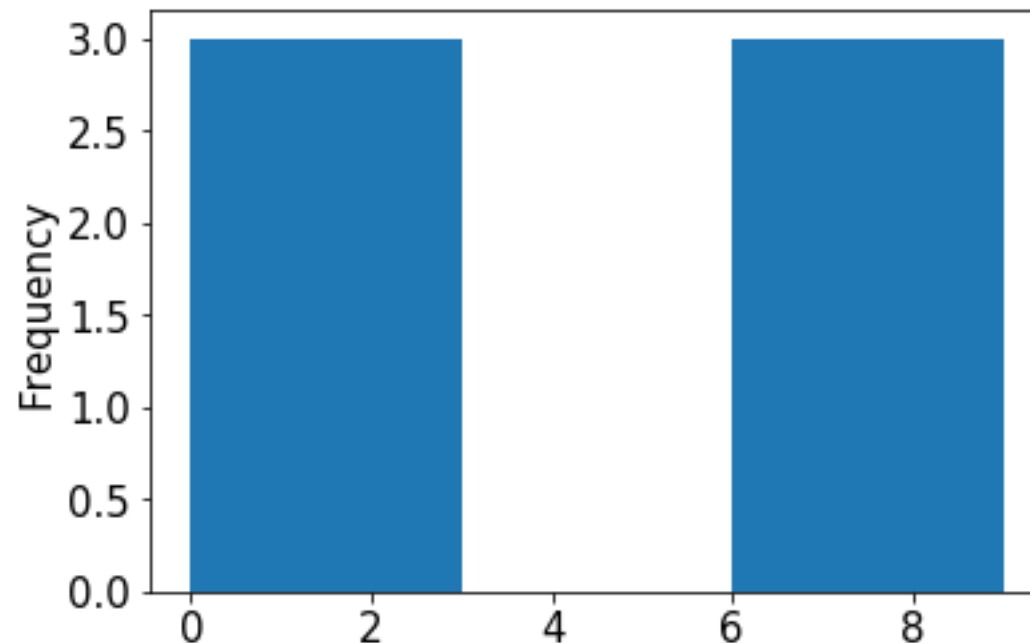
we can control the number of bins

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)
```



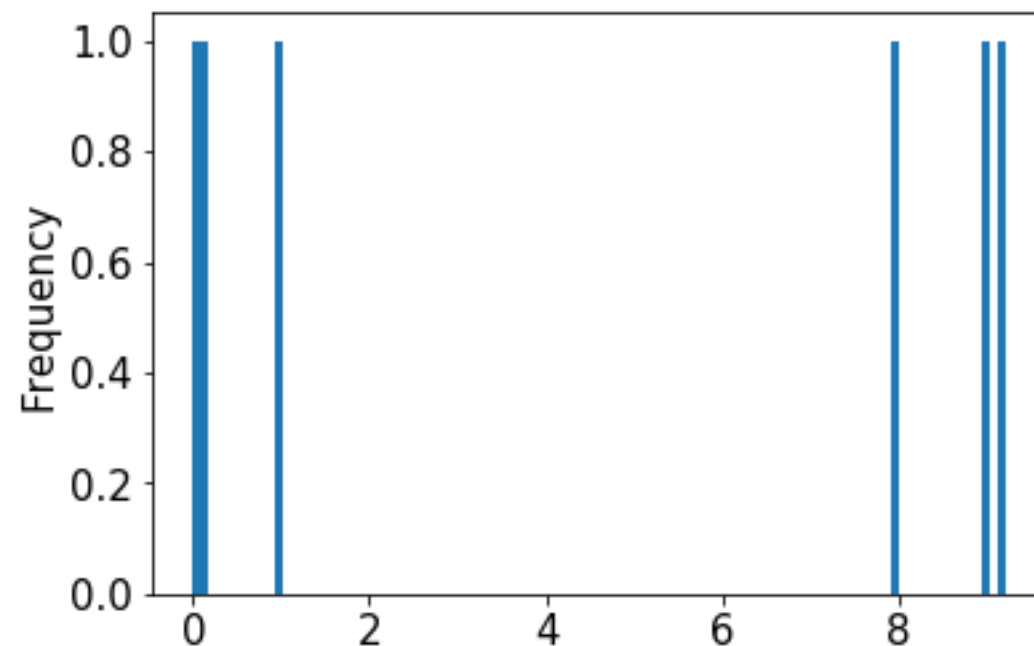
too few bins provides too little detail

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)
```



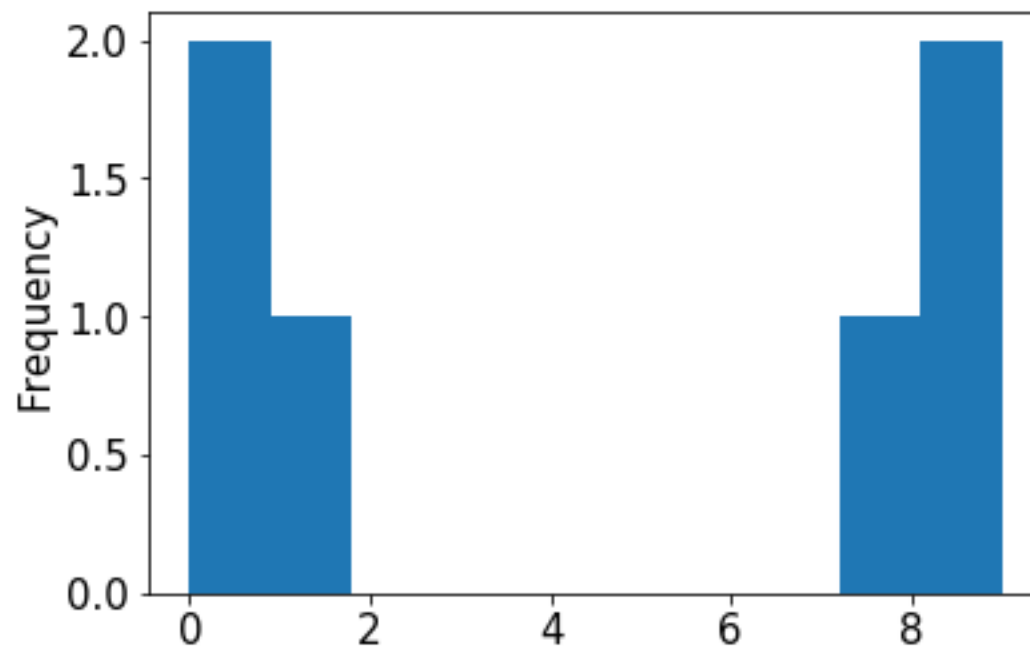
too many bins provides too much detail (equally bad)

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)
```



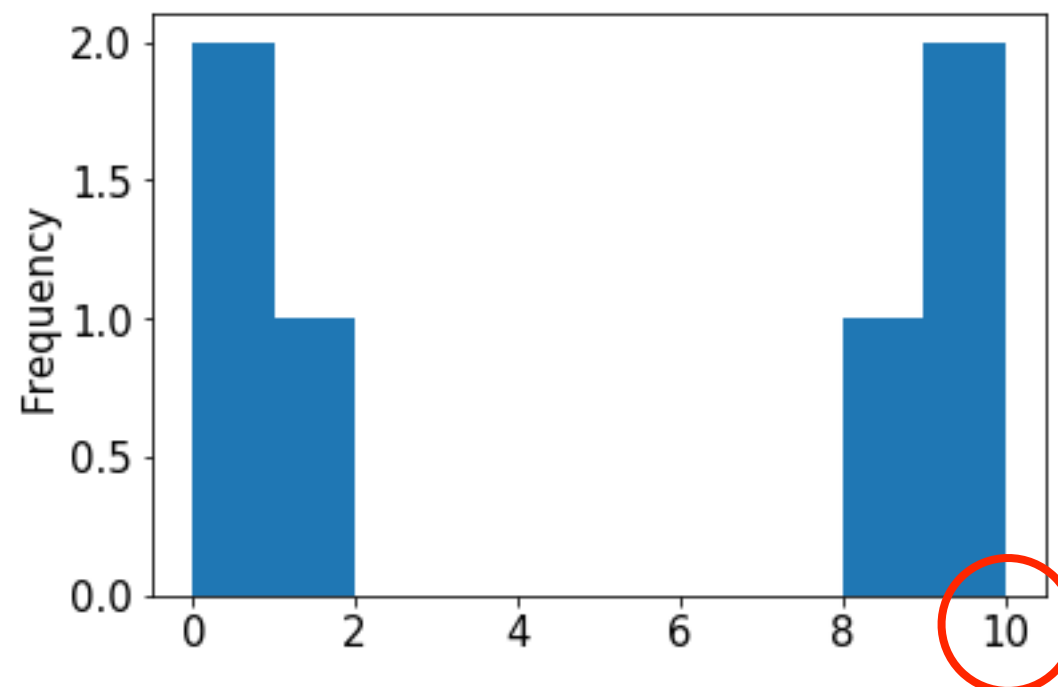
numpy chooses the default bin boundaries

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])
```



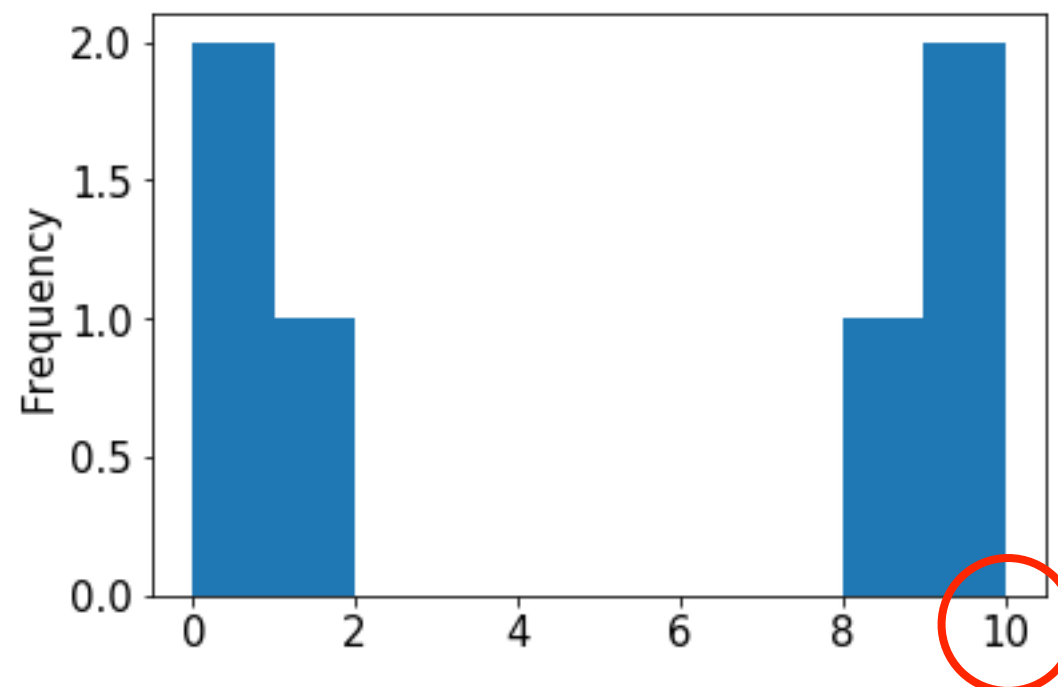
we can override the defaults

Frequencies across numbers

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

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

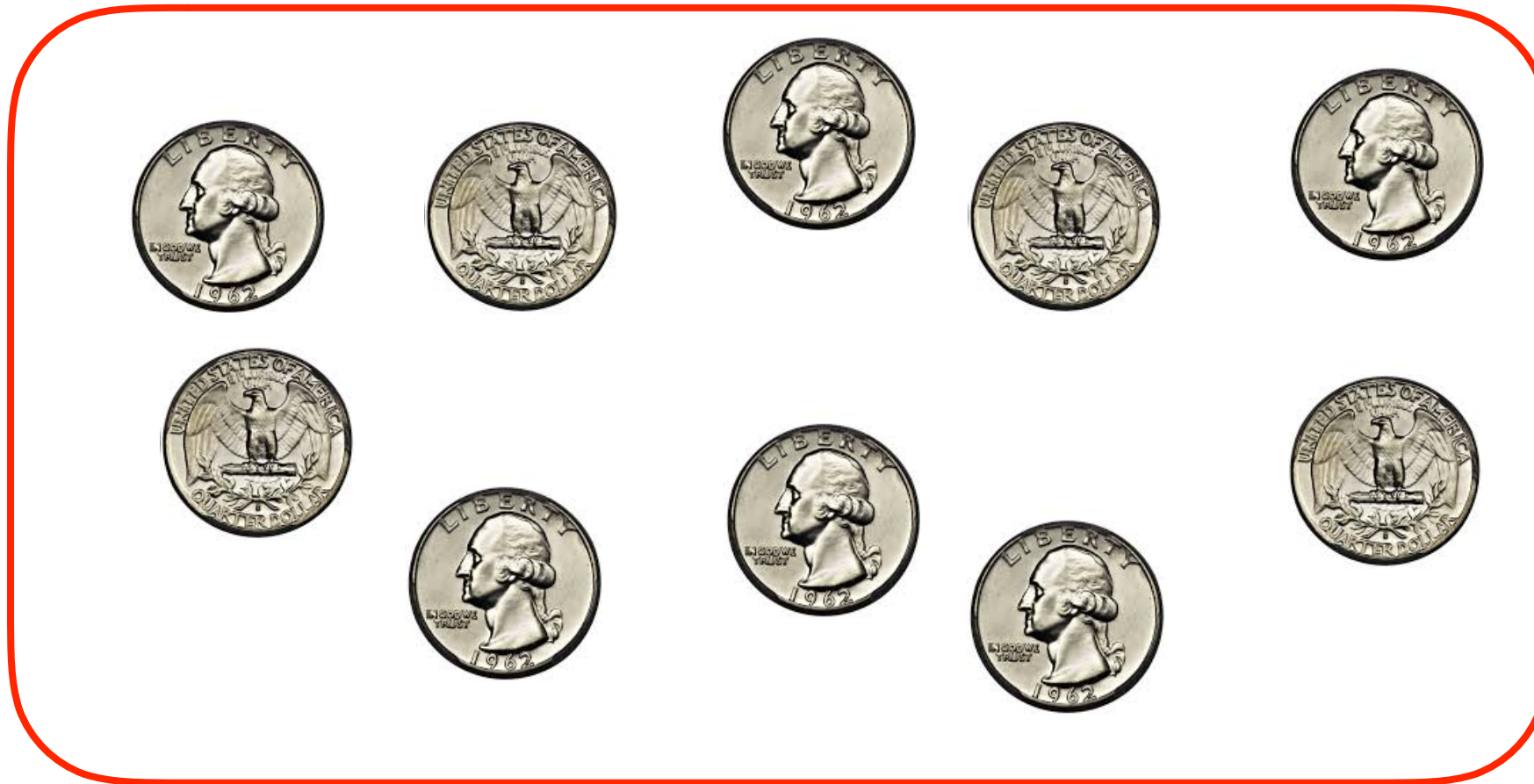
```
s.value_counts().sort_index().plot.bar()  
s.plot.hist(bins=range(11))
```



this is easily done with range

Demo 2: coin flips

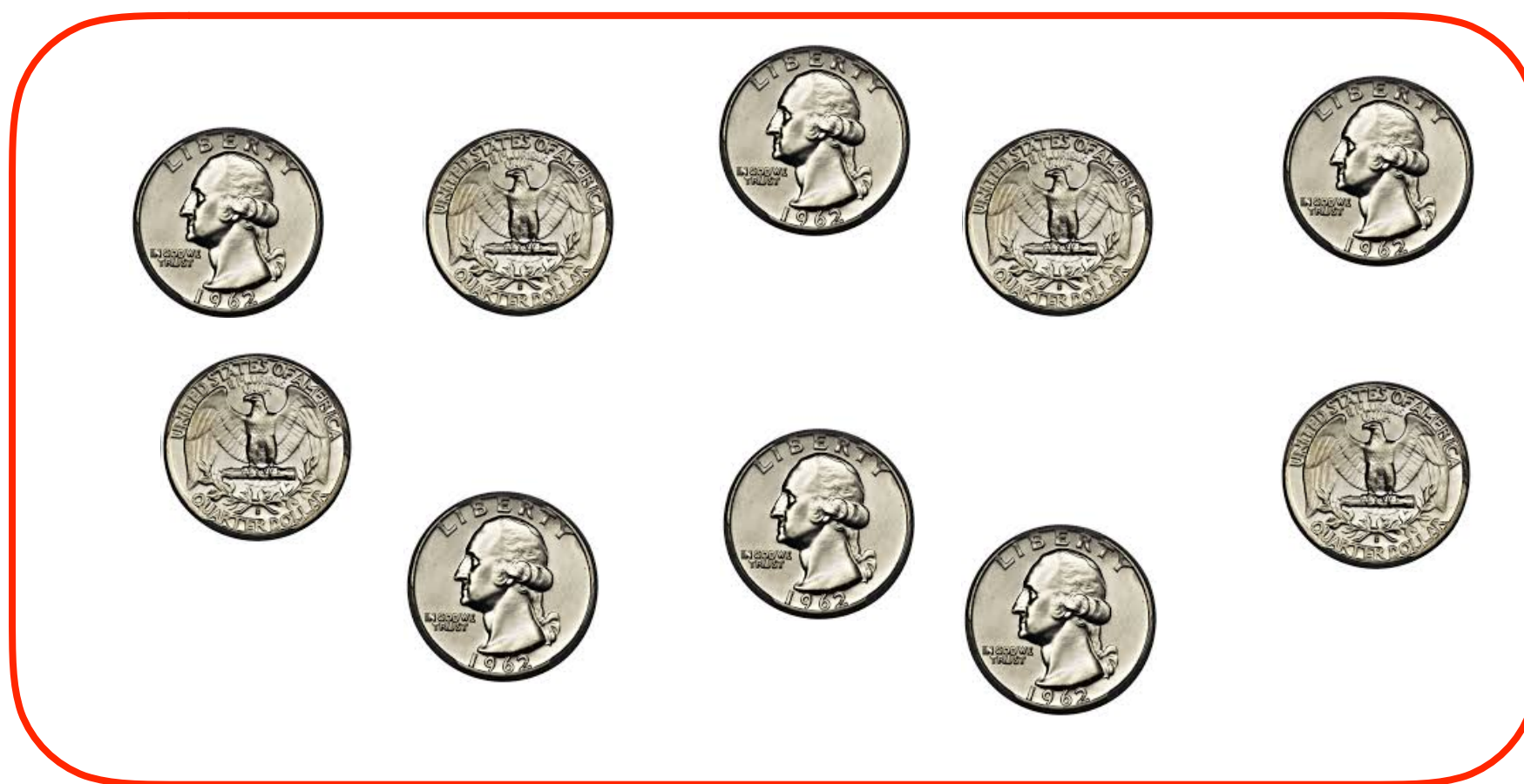
If we flip 10 coins repeatedly, we'll get varying numbers of heads



6 heads

Demo 2: coin flips

If we flip 10 coins repeatedly, we'll get varying numbers of heads



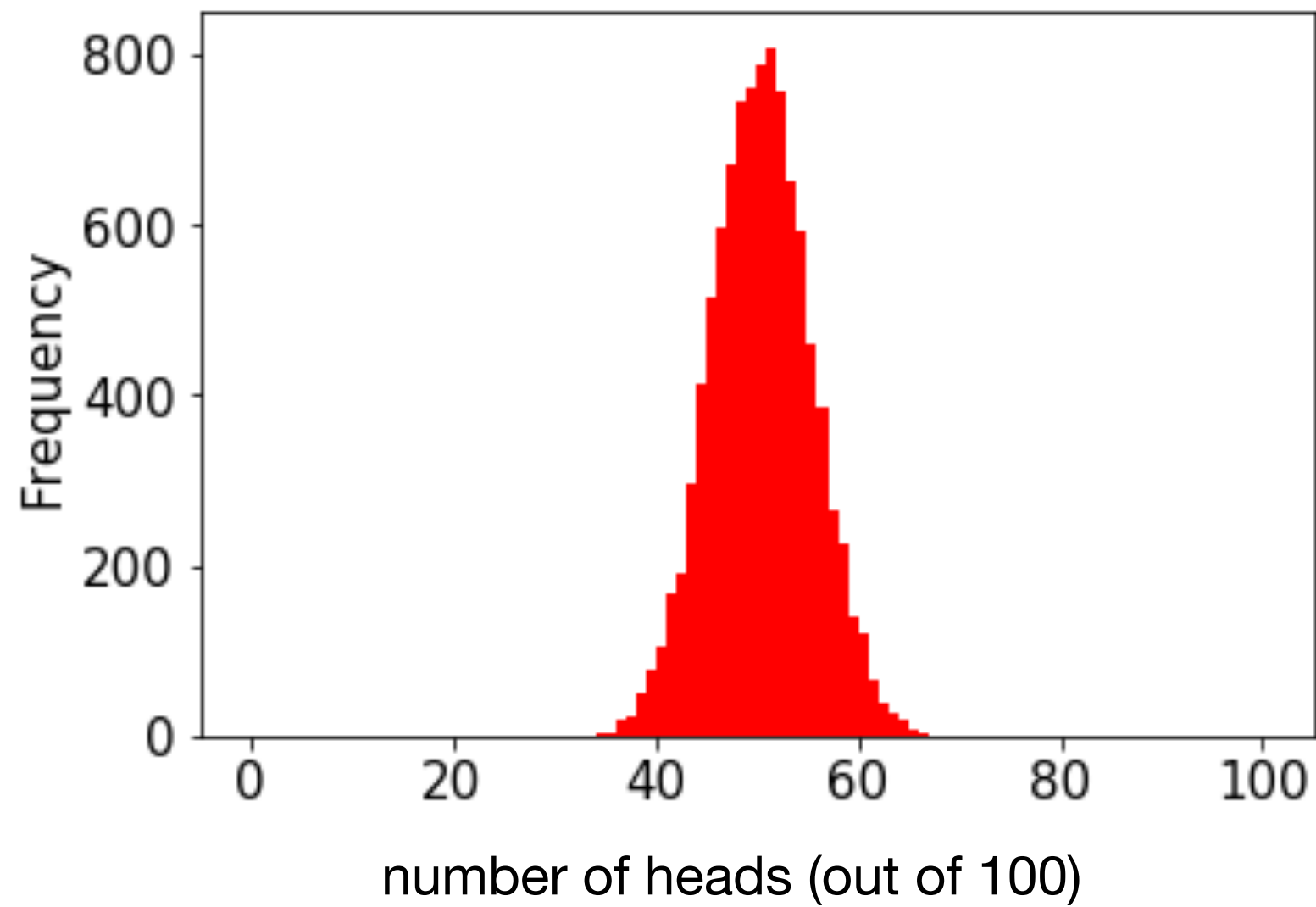
6 heads

If we flip 100 coins, 10K times, how often do we get each head count?

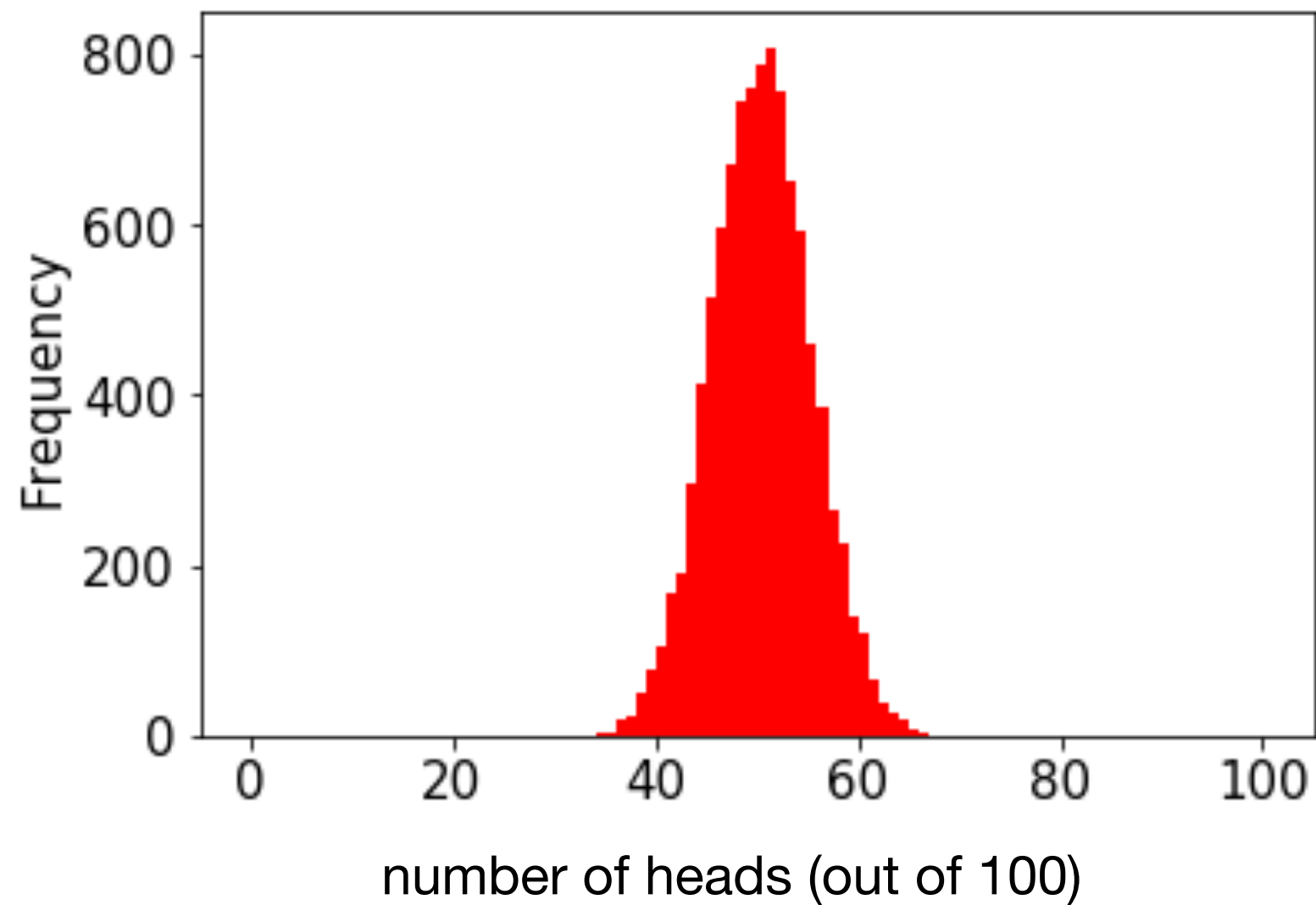
number of samples

sample size

Demo 2: result

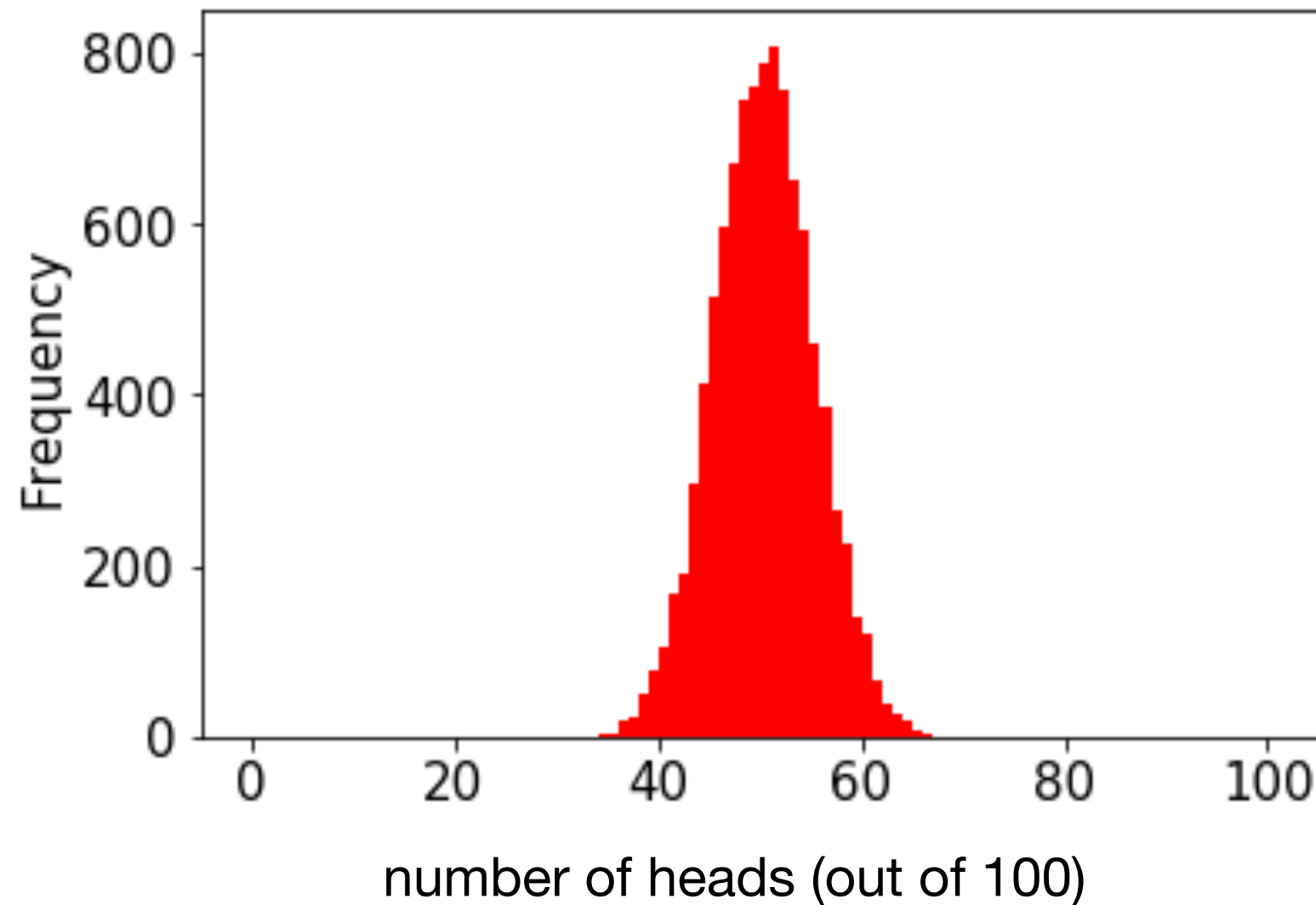


Demo 2: result



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

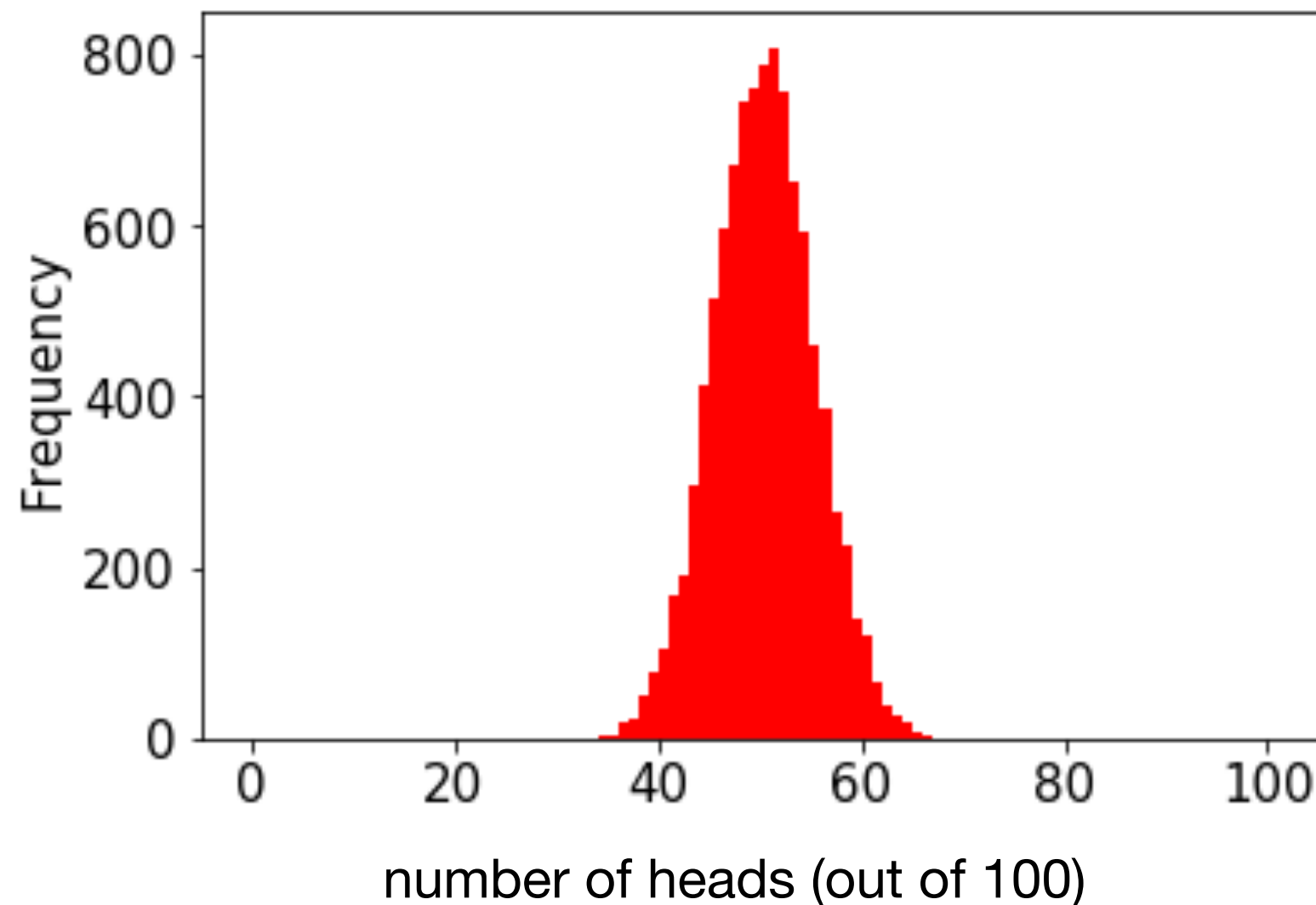
Demo 2: result



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

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

Demo 2: result



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 results will look like this
(we won't discuss exceptions here)

Outline

choice()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

statistical significance: an intuitive approach

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())
```

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
```

```
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))
```

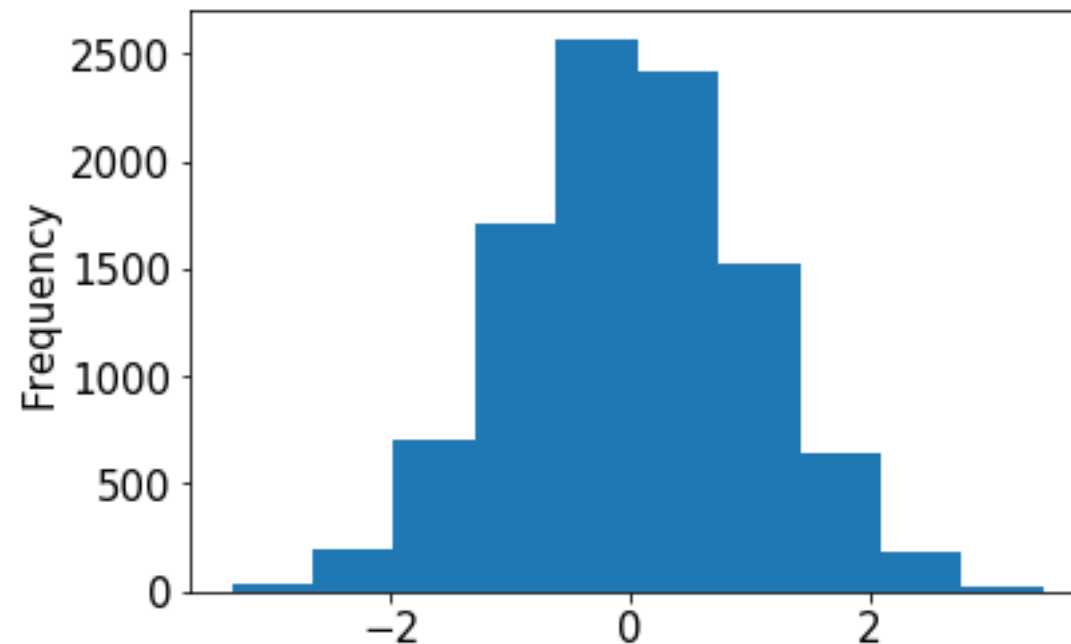
```
s.plot.hist()
```

normal

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

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

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

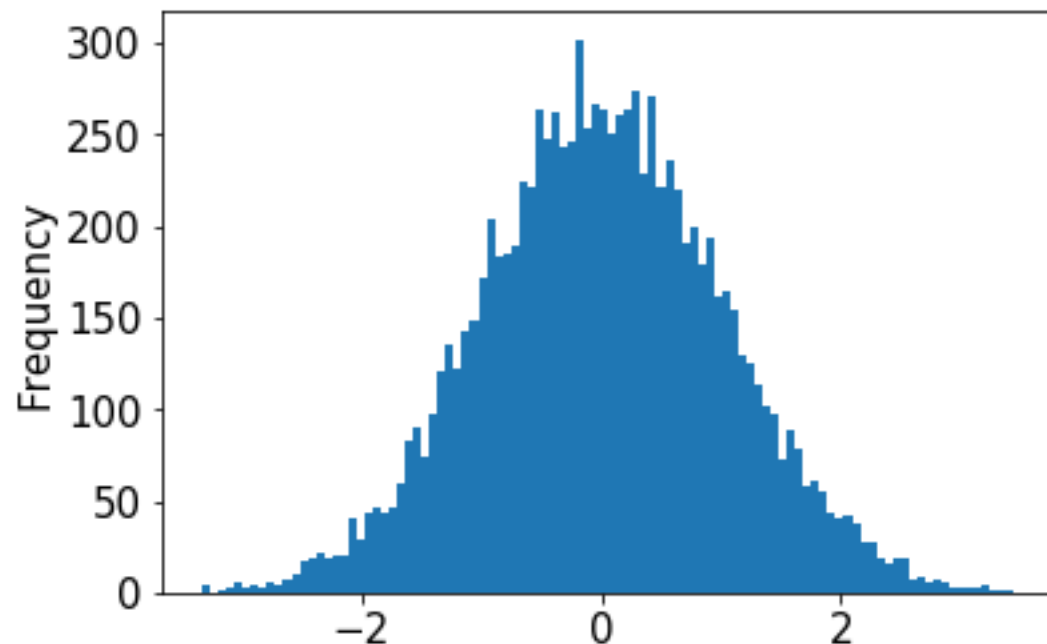


normal

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

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

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

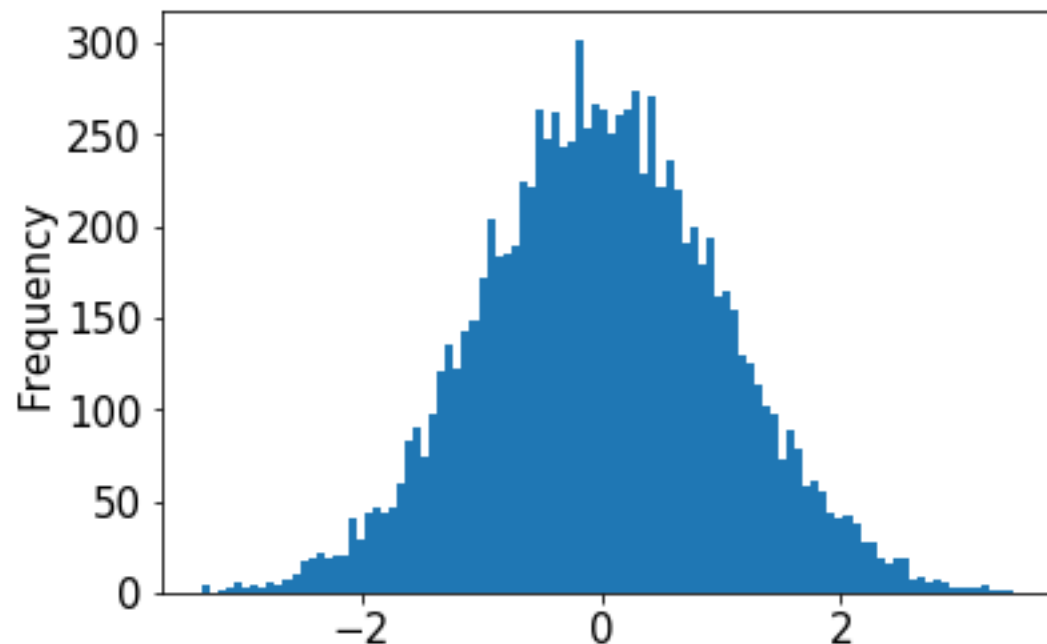


normal

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

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

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



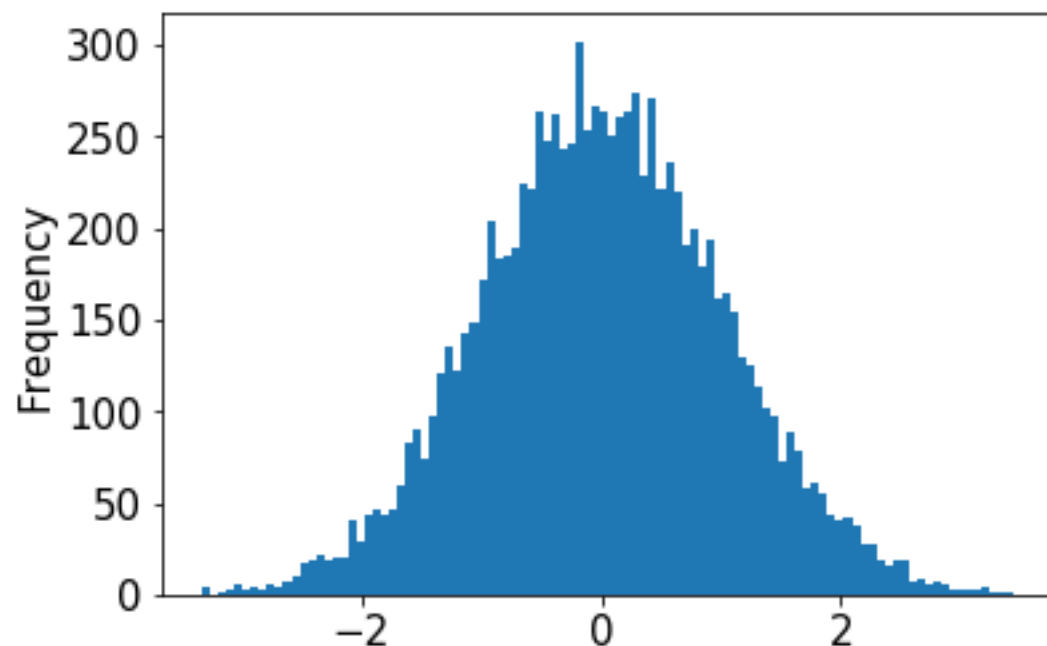
normal

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

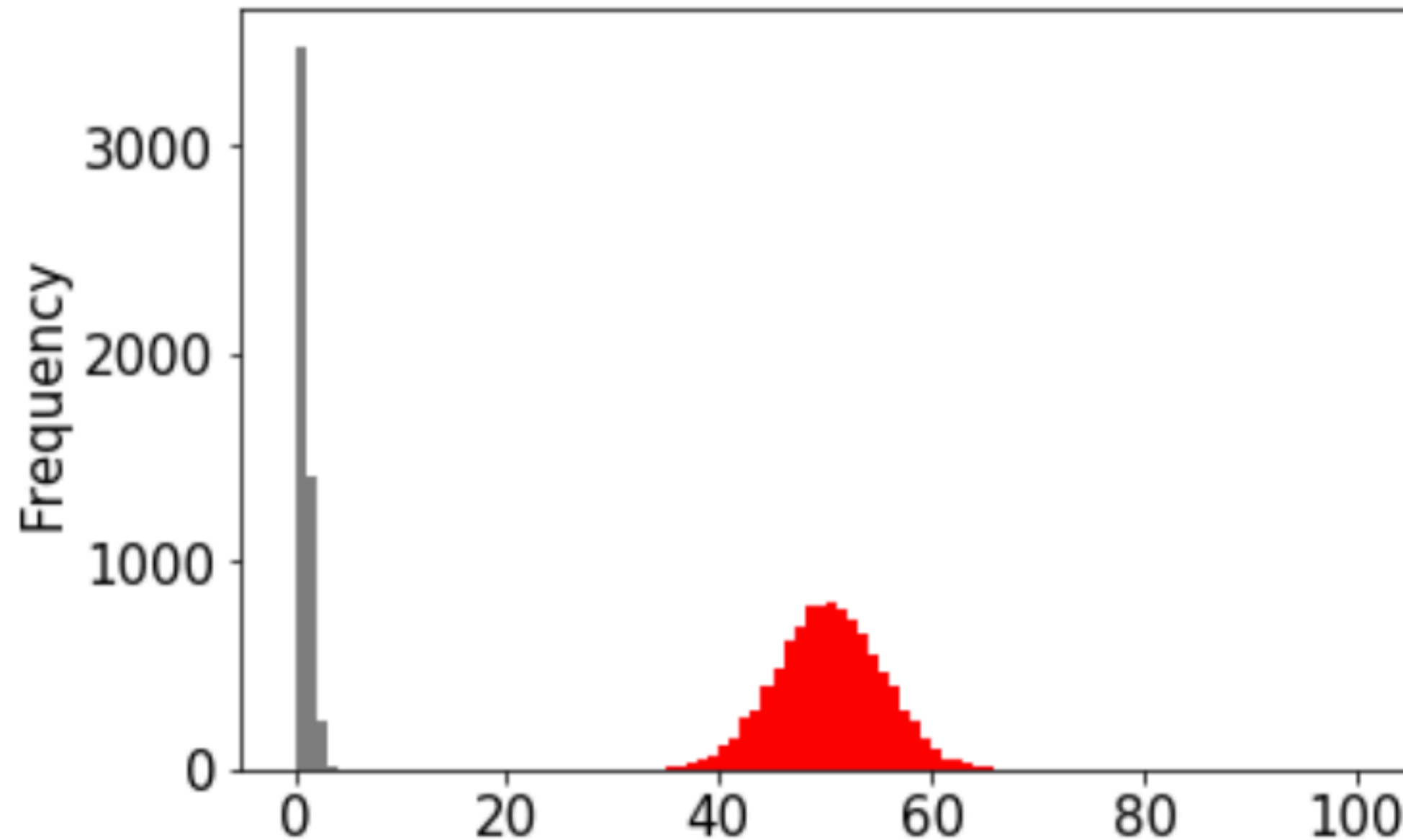
```
s = Series(normal(size=10000))
```

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

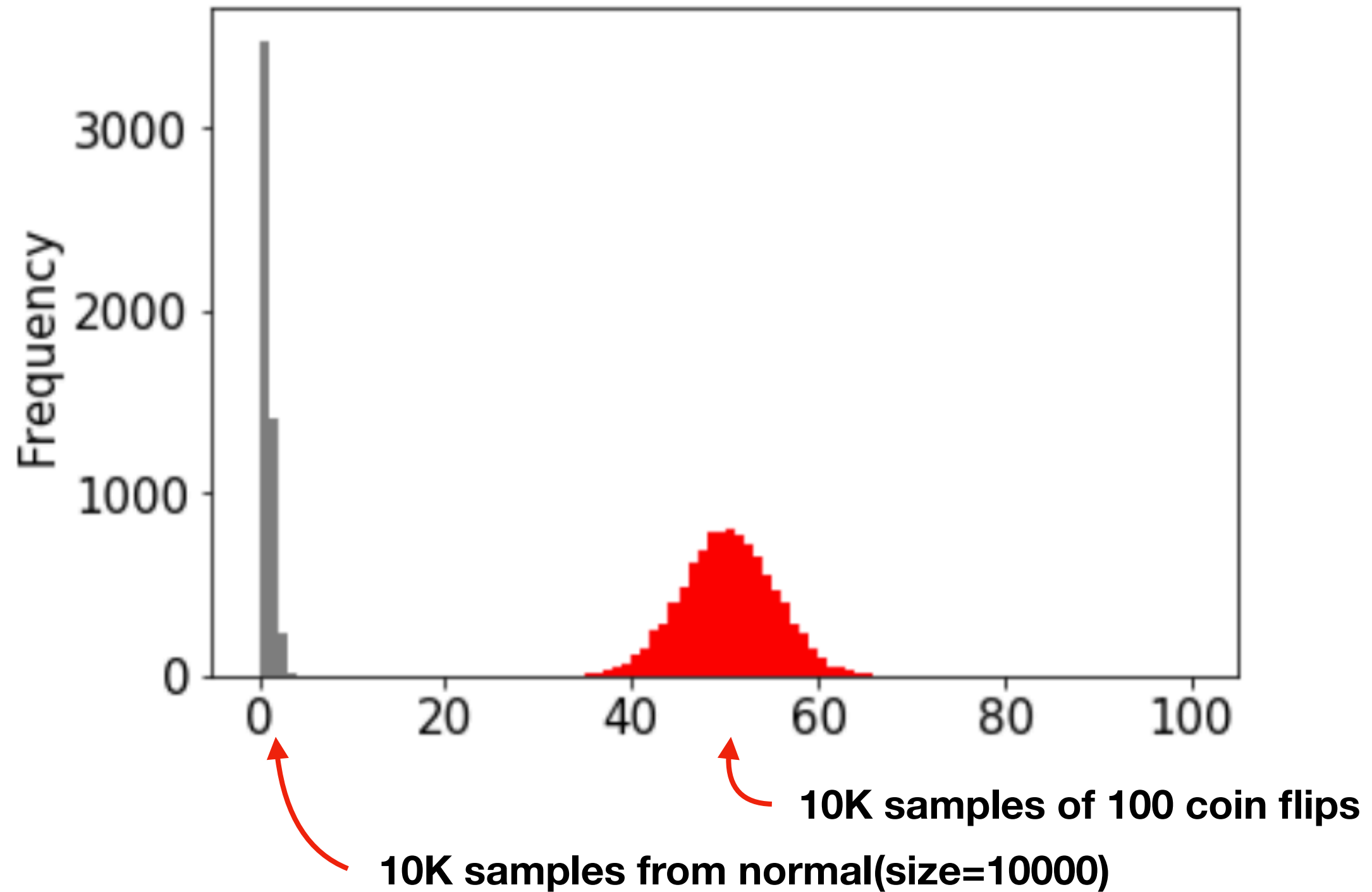
try plugging in different values
(defaults are 0 and 1, respectively)



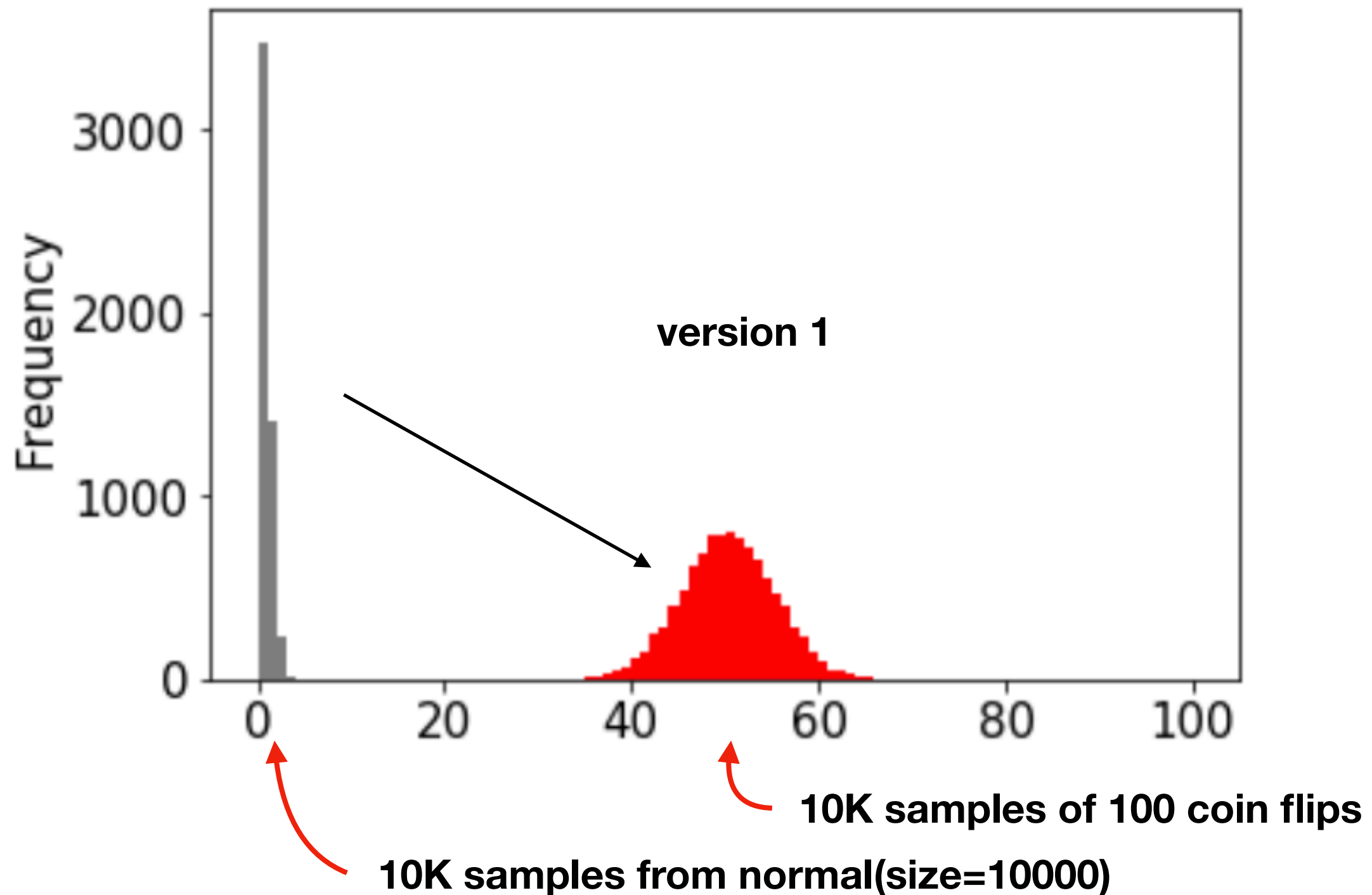
Demo 3: plot overlay



Demo 3: plot overlay

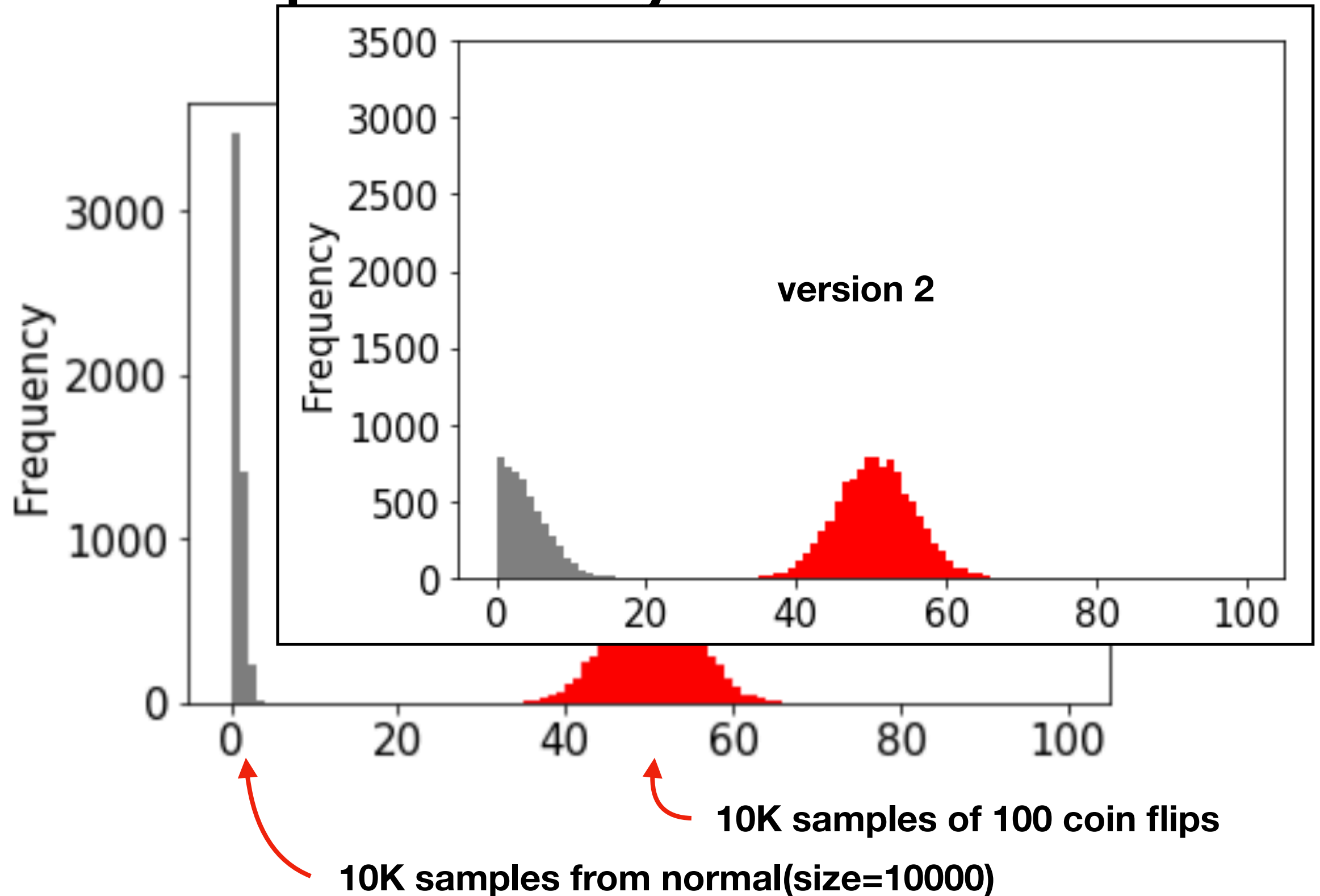


Demo 3: plot overlay



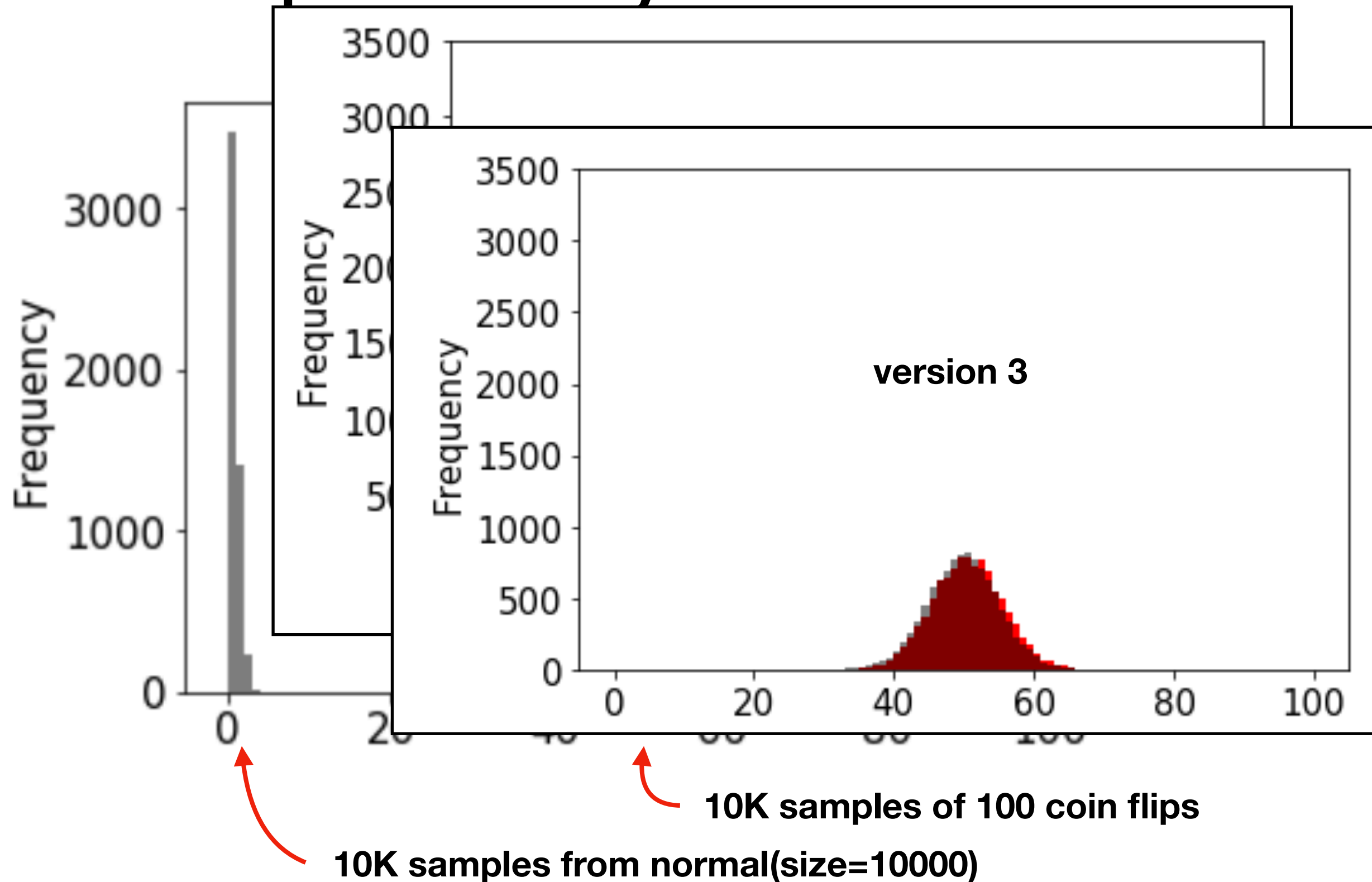
goal: play with **loc** and **scale** arguments to normal until gray overlaps red

Demo 3: plot overlay



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

Demo 3: plot overlay



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

Outline

choice()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

statistical significance: an intuitive approach

Is this coin biased?

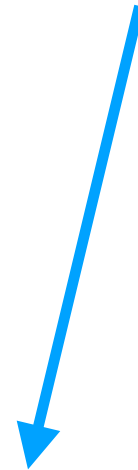


51



49

Call shenanigans?



TOP DEFINITION



call shenanigans

v. to **declare** that another's words or behavior is full of shit, **off topic**, or passive-aggressively annoying. to call another on their bad or **mischievous** behavior.

*"**Will you** stop prevaricating and answer **the question**? **I call shenanigans!**"*

#shenanigans #deception #bullshit #passive-aggresssive #mischievious

by **Chri5tina** October 09, 2008



105



21



<https://www.urbandictionary.com/define.php?term=call%20shenanigans>

Is this coin biased?



51




49

Call shenanigans?

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

TOP DEFINITION




call shenanigans

v. to **declare** that another's words or behavior is full of shit, **off topic**, or passive-aggressively annoying. to call another on their bad or **mischievous** behavior.

*"**Will you** stop prevaricating and answer **the question**? **I call shenanigans!**"*

#shenanigans #deception #bullshit #passive-aggressive #mischievous

by **Chri5tina** October 09, 2008

 105  21



<https://www.urbandictionary.com/define.php?term=call%20shenanigans>

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.

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?



55 million



45 million

Call shenanigans?

Is this coin biased?



51



49

Call shenanigans? No.



5



95

Call shenanigans? Yes.



55



45

Call shenanigans? No.



55 million



45 million

Call shenanigans? Yes.

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 4: Calling Shenanigans



60



40

Call shenanigans?

Demo 4: Calling Shenanigans



60



40

Call shenanigans?

Strategy: simulate a fair coin

Demo 4: Calling Shenanigans



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

Demo 4: Calling Shenanigans



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 4: Calling Shenanigans



60



40

Call shenanigans?

we got 10 more heads than we expect on average

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 4: Calling Shenanigans



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 4: Calling Shenanigans



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

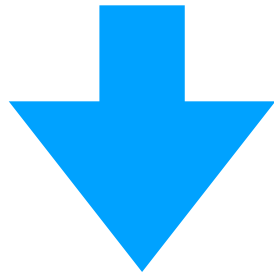
[50, 61, 51, 44, 39, 43, 51, 49, 49, 38, ...]

11 more

12 less

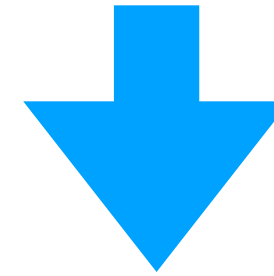
Demo 5: Do front-row students score better?

```
df = pd.read_csv("scores.csv")
```



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|-----|-----|----|----|----|----|----|-----|-----|----|
| 0 | 84 | 90 | 89 | 89 | 90 | 87 | 85 | 100 | 96 | 94 |
| 1 | 100 | 89 | 89 | 87 | 72 | 88 | 72 | 98 | 94 | 82 |
| 2 | 85 | 82 | 83 | 84 | 73 | 85 | 76 | 94 | 87 | 96 |
| 3 | 74 | 87 | 82 | 89 | 97 | 91 | 84 | 80 | 91 | 87 |
| 4 | 98 | 76 | 78 | 77 | 91 | 88 | 78 | 100 | 77 | 70 |
| 5 | 82 | 72 | 73 | 84 | 98 | 70 | 94 | 91 | 73 | 83 |
| 6 | 70 | 100 | 94 | 76 | 71 | 75 | 71 | 77 | 100 | 73 |
| 7 | 89 | 77 | 83 | 71 | 95 | 89 | 77 | 92 | 91 | 70 |
| 8 | 100 | 84 | 82 | 79 | 70 | 88 | 98 | 77 | 81 | 79 |
| 9 | 99 | 88 | 89 | 92 | 84 | 82 | 94 | 93 | 77 | 75 |

```
df.mean(axis=1)
```



| | |
|----------------|------|
| 0 | 90.4 |
| 1 | 87.1 |
| 2 | 84.5 |
| 3 | 86.2 |
| 4 | 83.3 |
| 5 | 82.0 |
| 6 | 80.7 |
| 7 | 83.4 |
| 8 | 83.8 |
| 9 | 87.3 |
| dtype: float64 | |

what are the odds that the front row would do so well by chance?