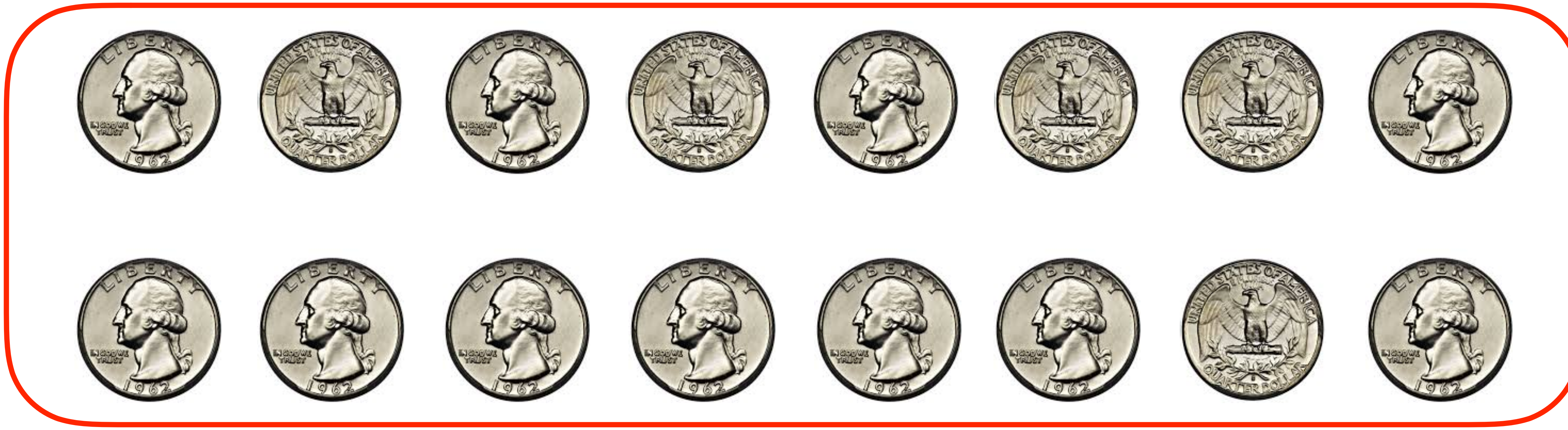
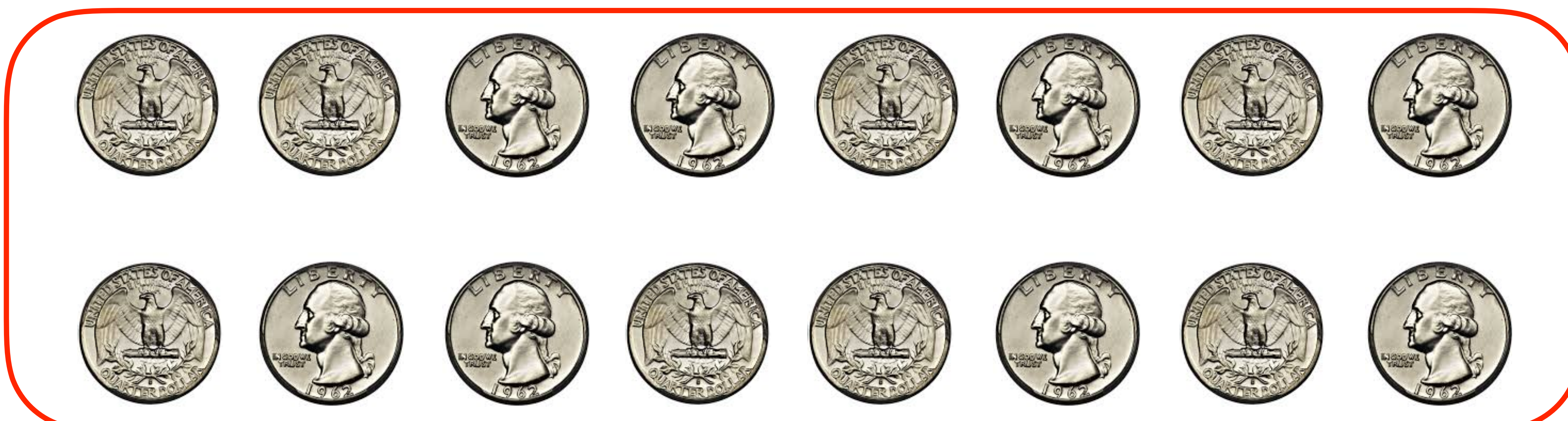
[301] Randomness

Tyler Caraza-Harter

Which series was randomly generated?
Which did I pick by hand?





TODO

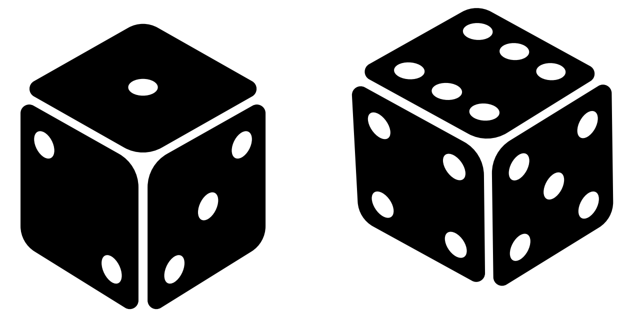
winter-break reading

wed review session: how to prep

Why Randomize?

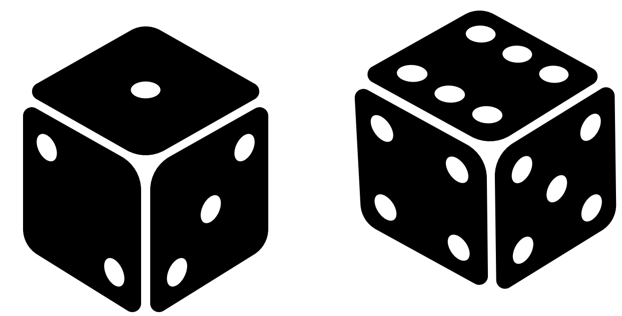
Why Randomize?

Games



Why Randomize?

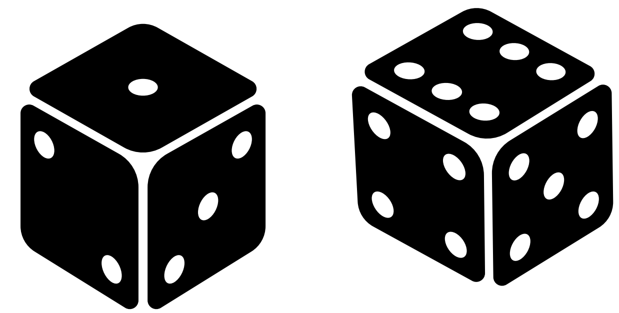
Games



Security

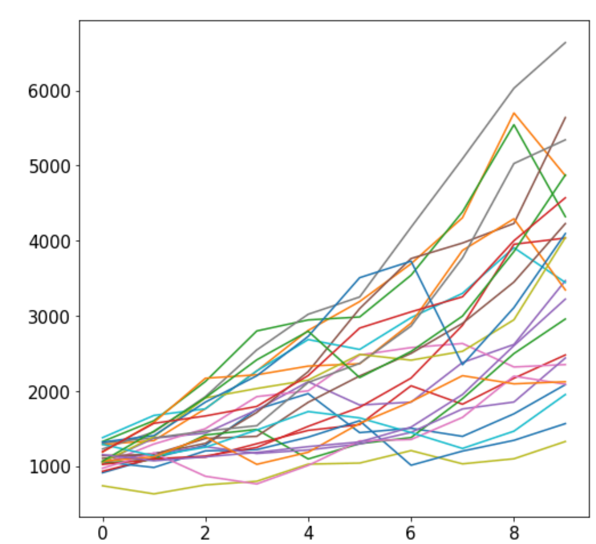
Why Randomize?

Games



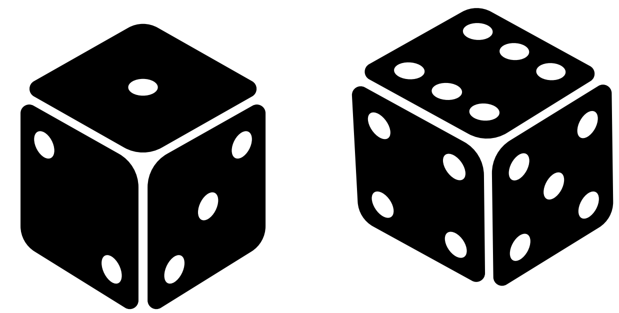
Security

Simulation



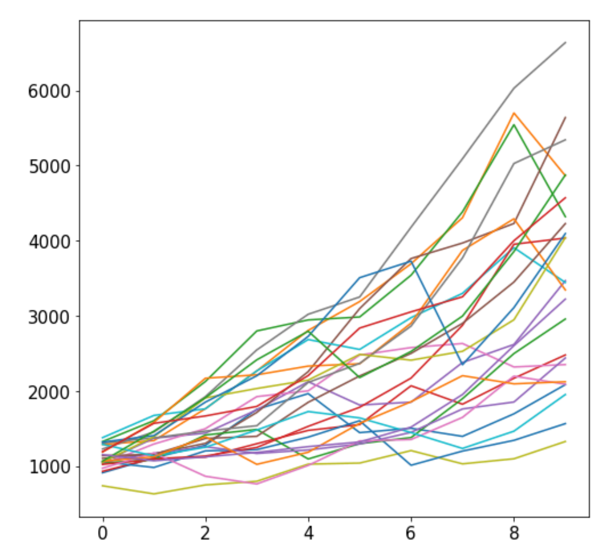
Why Randomize?

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`**

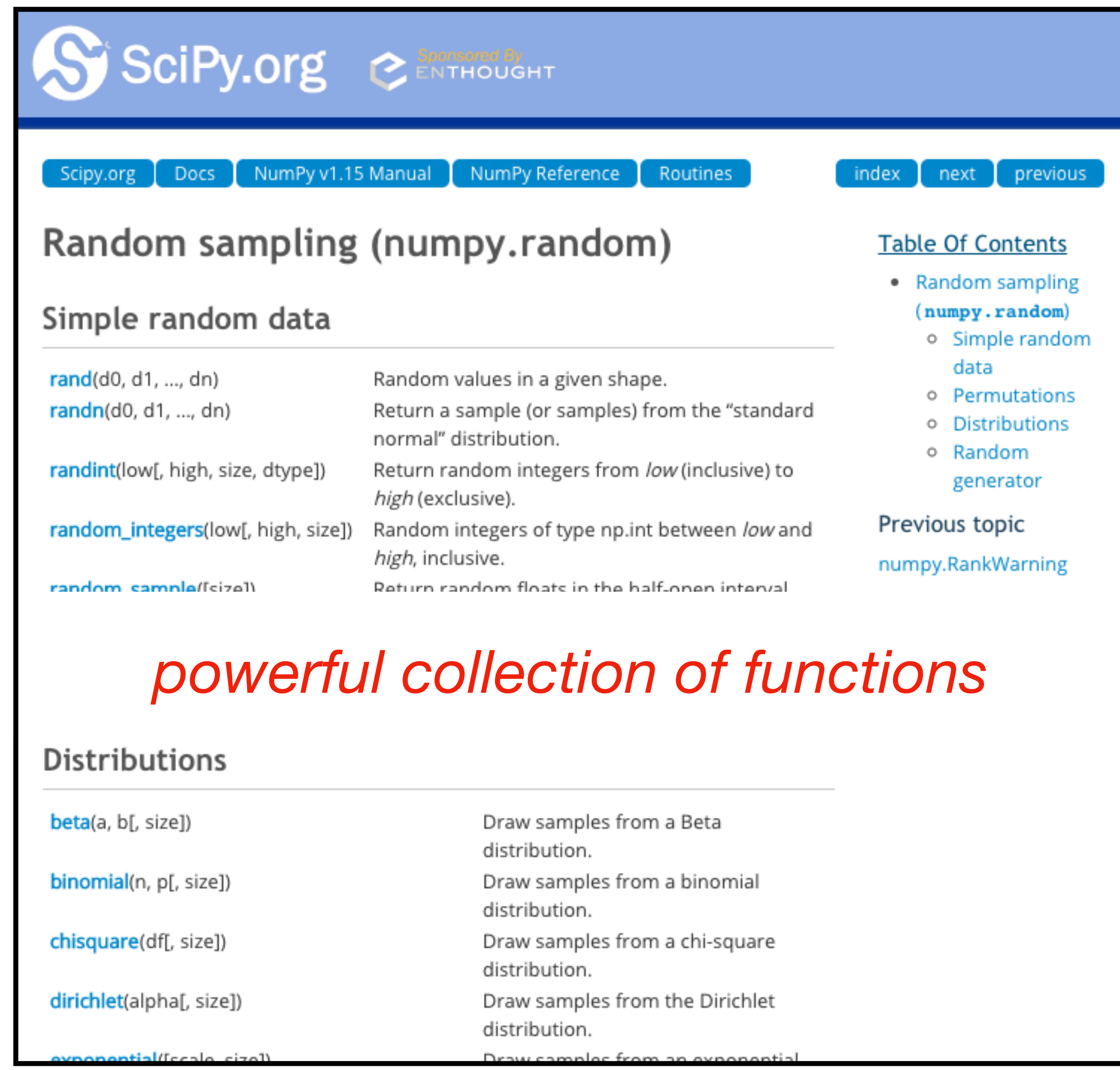
New Functions Today

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

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

numpy.random:

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



New Functions Today

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

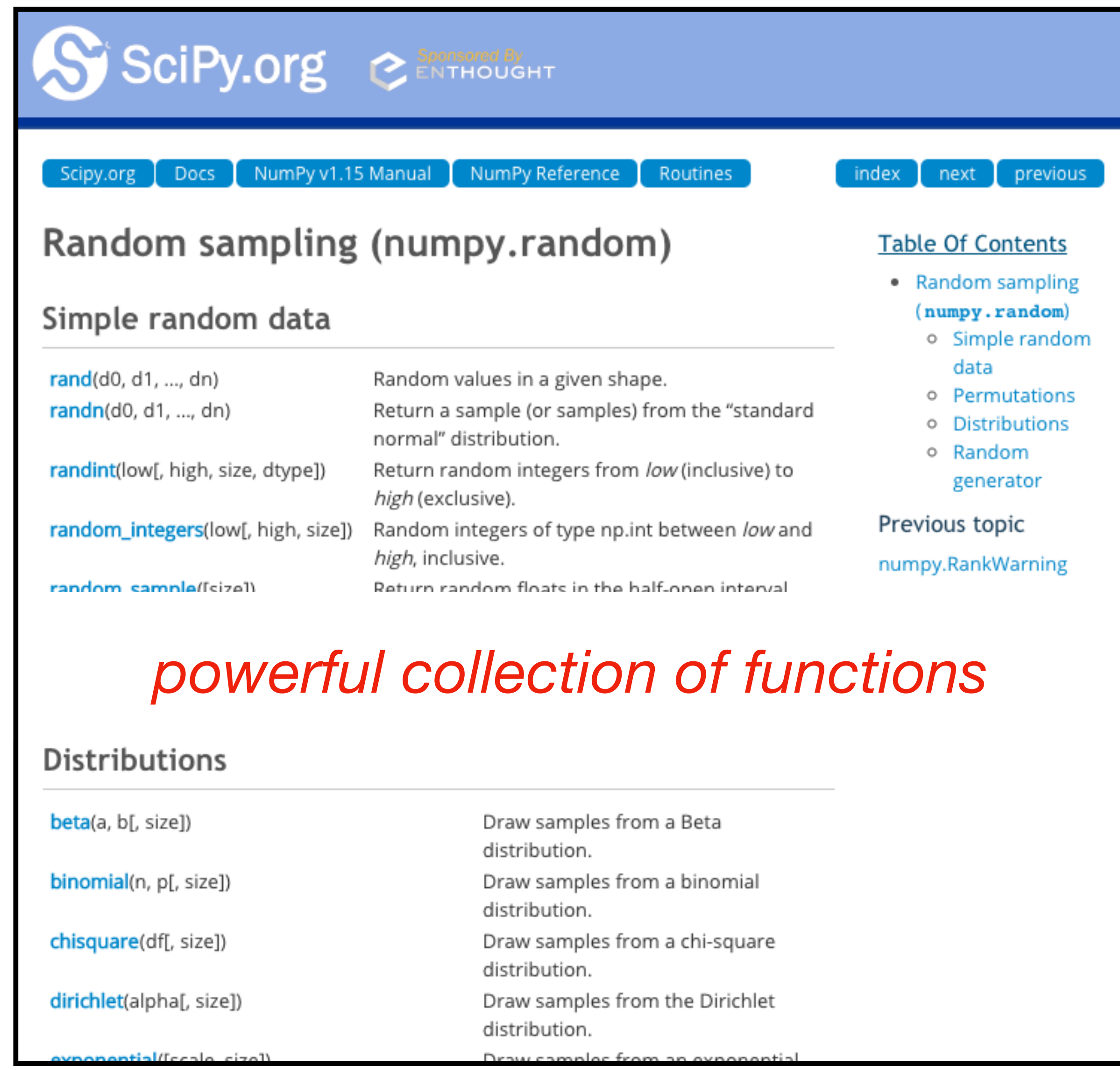
- **`choice, choices, randint`**

numpy.random:

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

`Series.line.hist`:

- similar to bar plot
- visualize spread of random results



New Functions Today

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

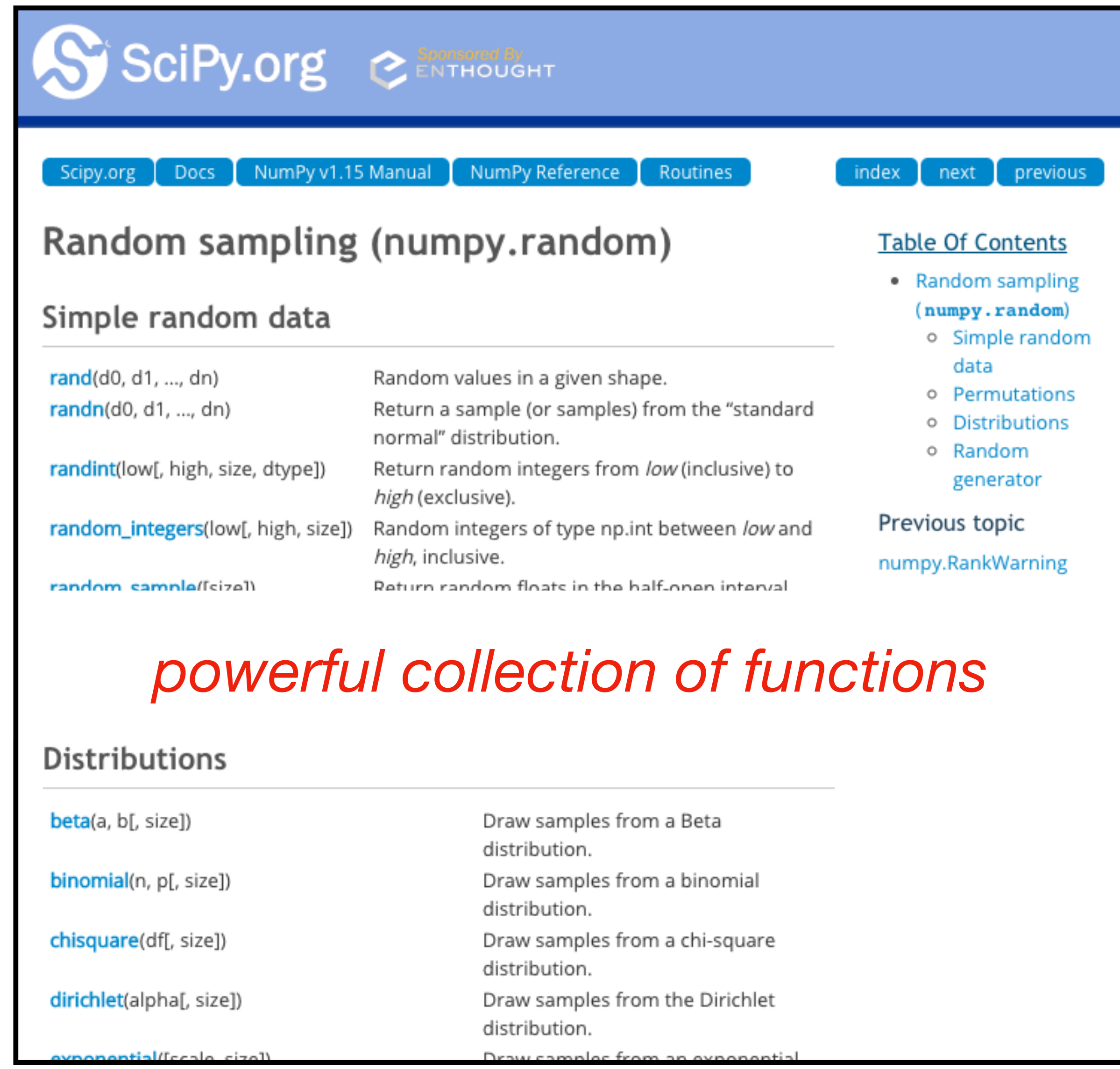
- **`choice, choices, randint`**

numpy.random:

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

`Series.line.hist`:

- similar to bar plot
- visualize spread of random results



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

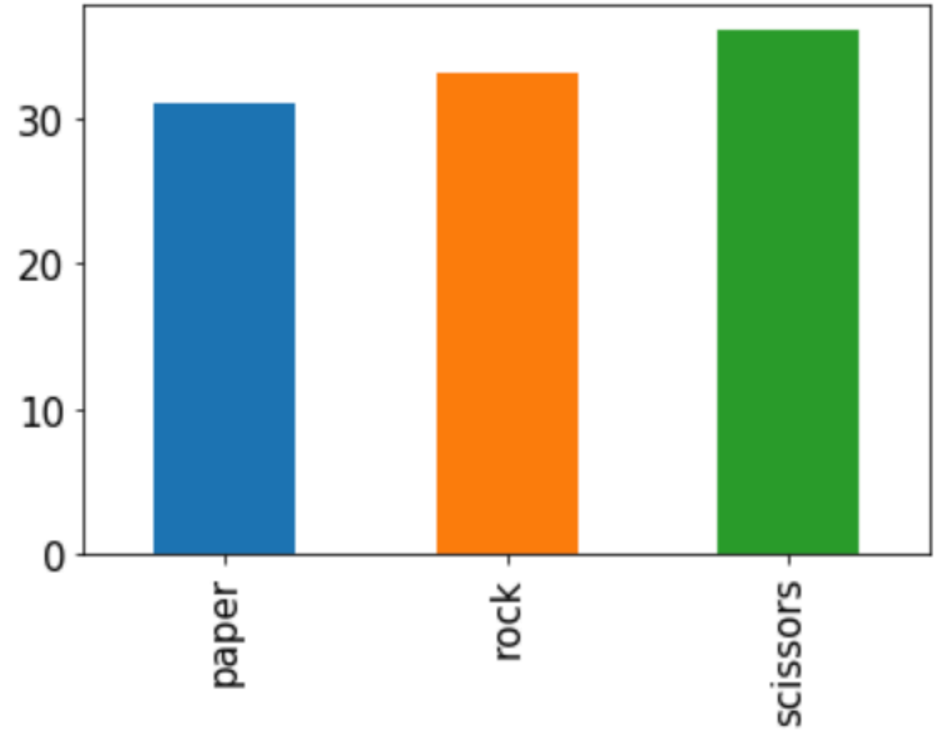
```
choice(["rock", "paper", "scissors"])
```

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?

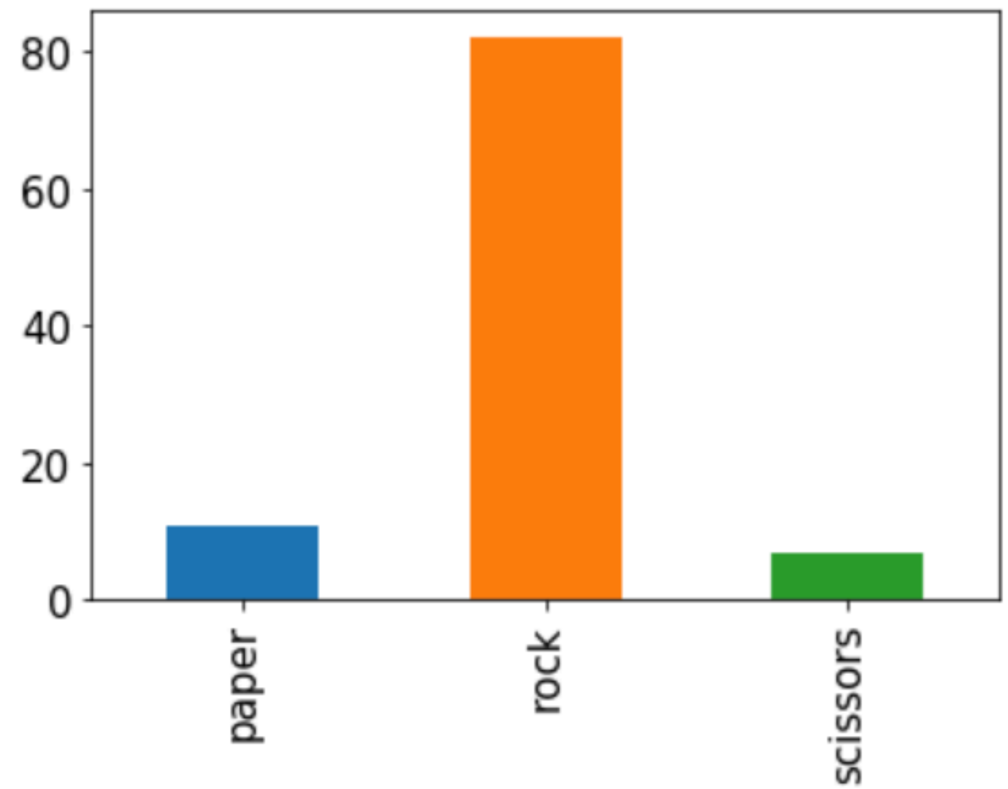


Demo 1: exploring bias

```
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])
```

same

```
# 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])
```

same

```
# 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))
```

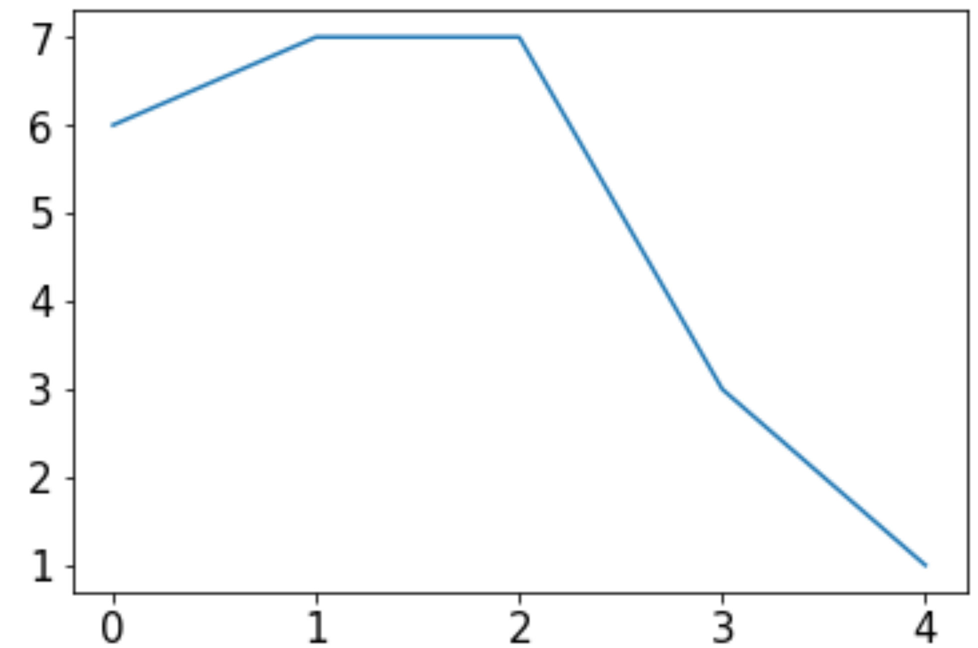
```
0    6
1    7
2    7
3    3
4    1
dtype: int64
```

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()
```

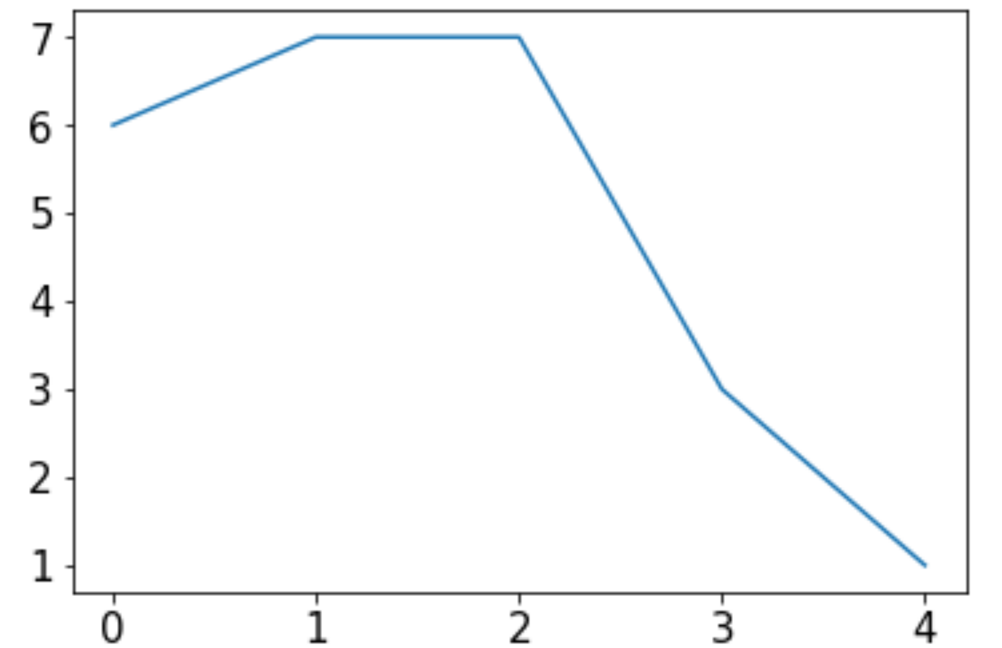


Example: change over time

```
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?

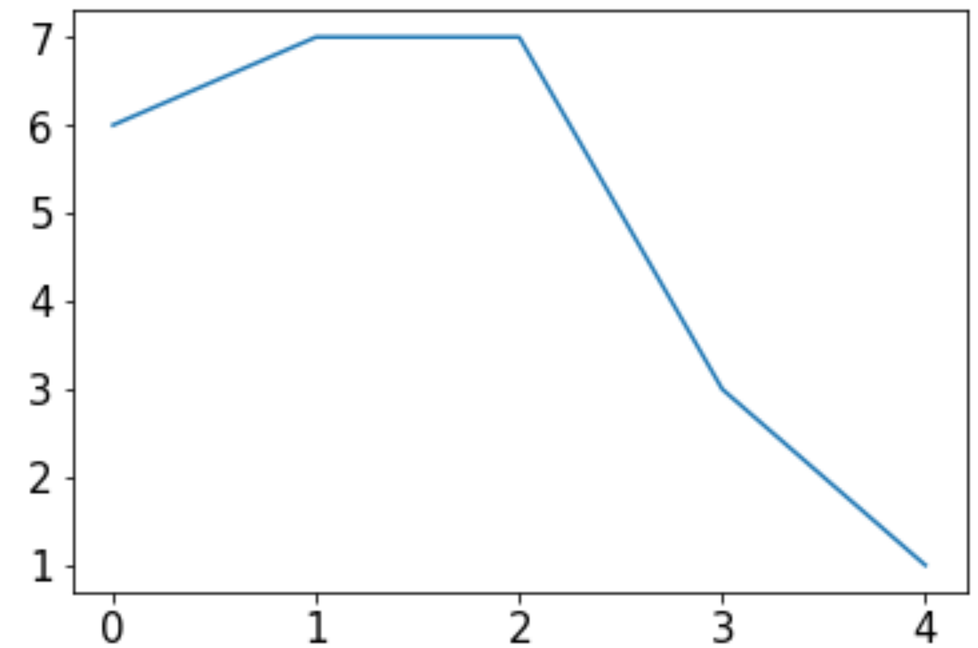
Example: change over time

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

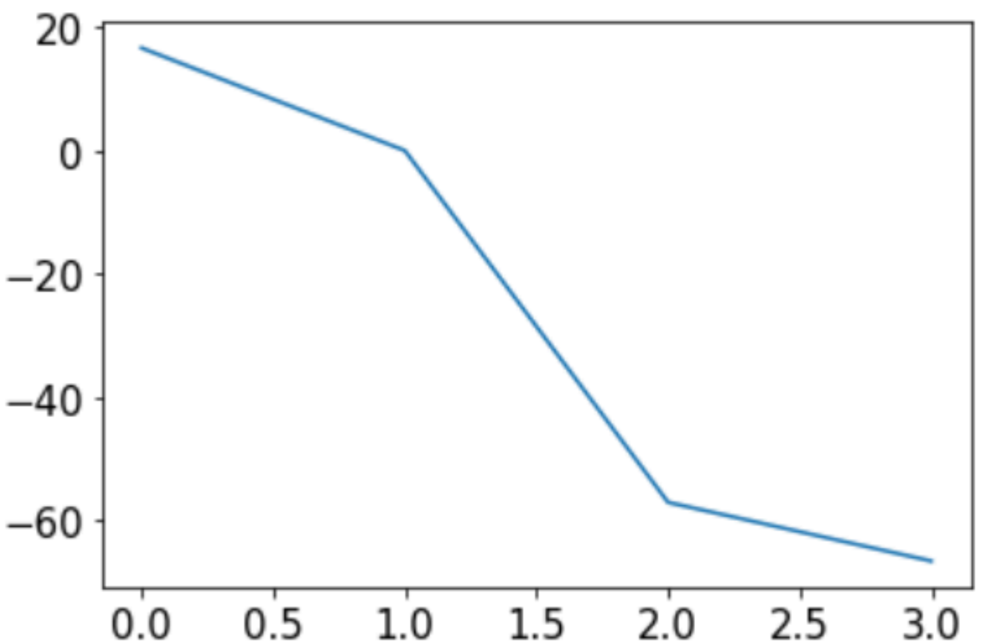
0	6
1	7
2	7
3	3
4	1

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()
```

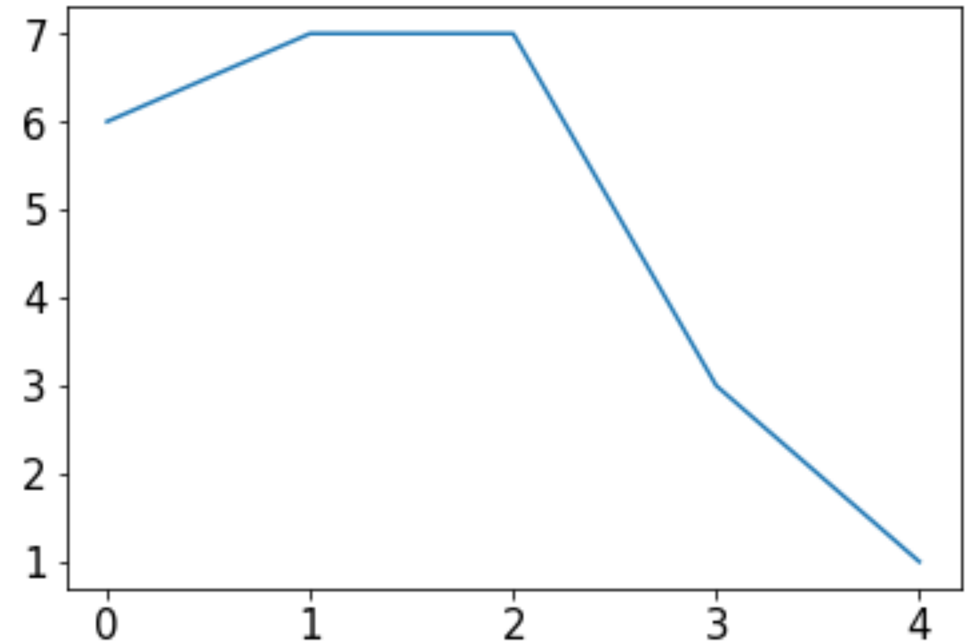


Example: change over time

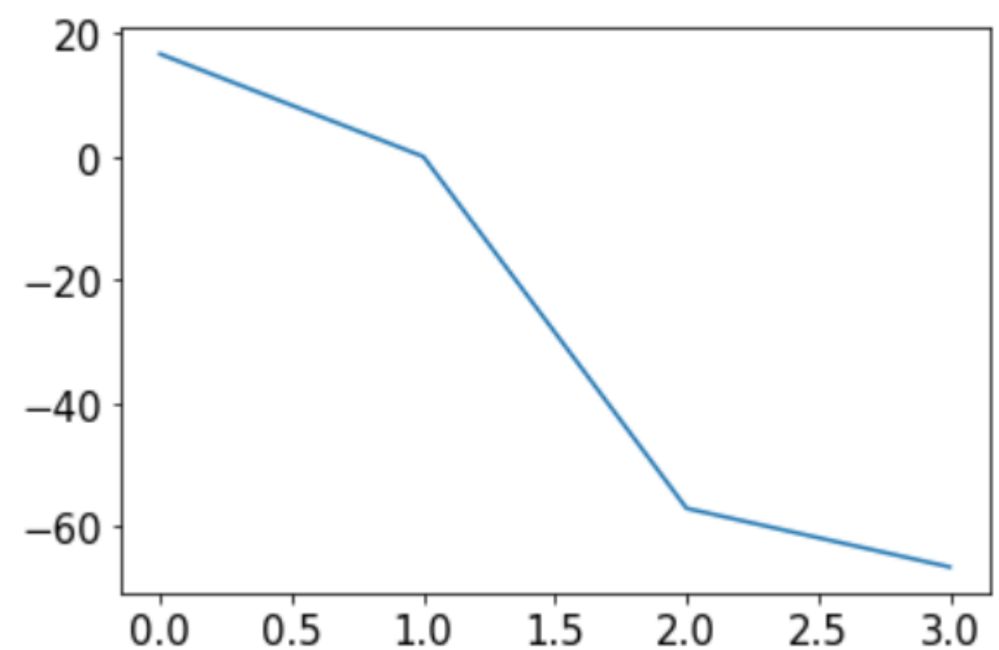
```
s = Series(choice(10, size=5))
```

```
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()
```

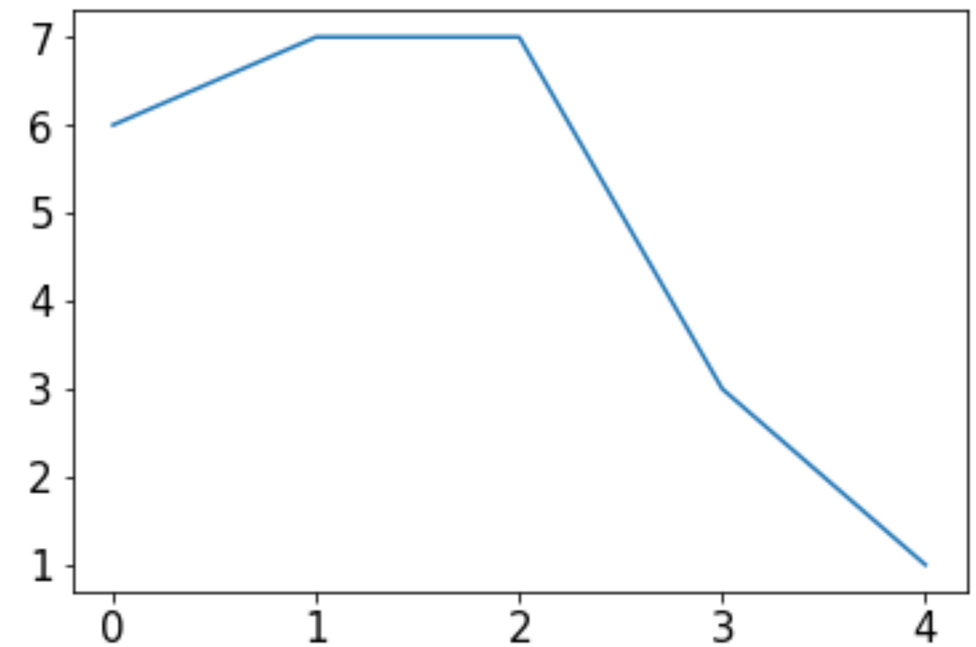


can you identify the bug in the code?

Example: change over time

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

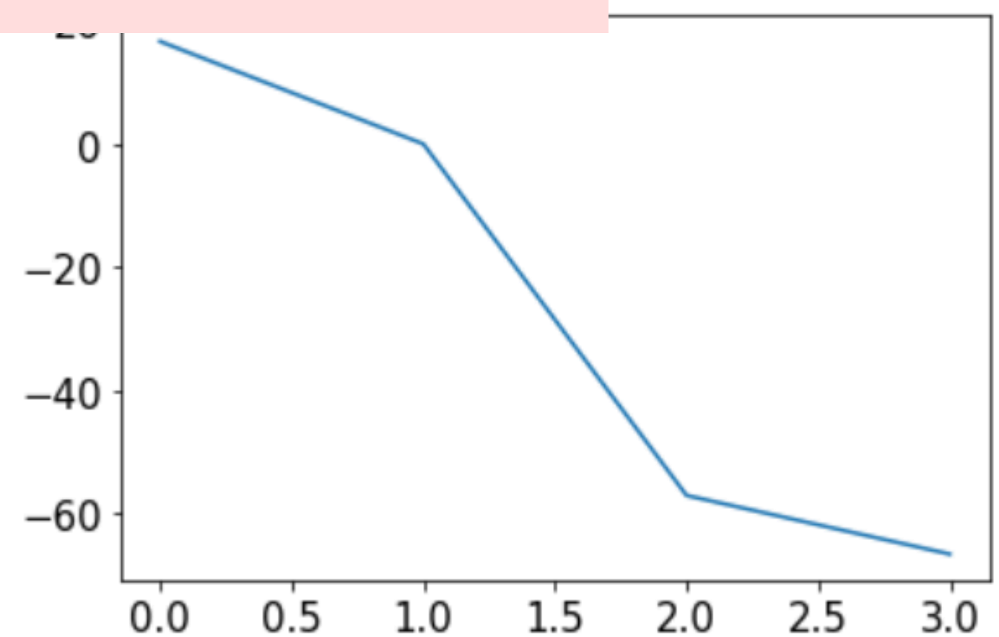
```
0      6
1      7
2      7
3      3
4      1
dtype: int64
```




```
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, $\boxed{249}$, 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()

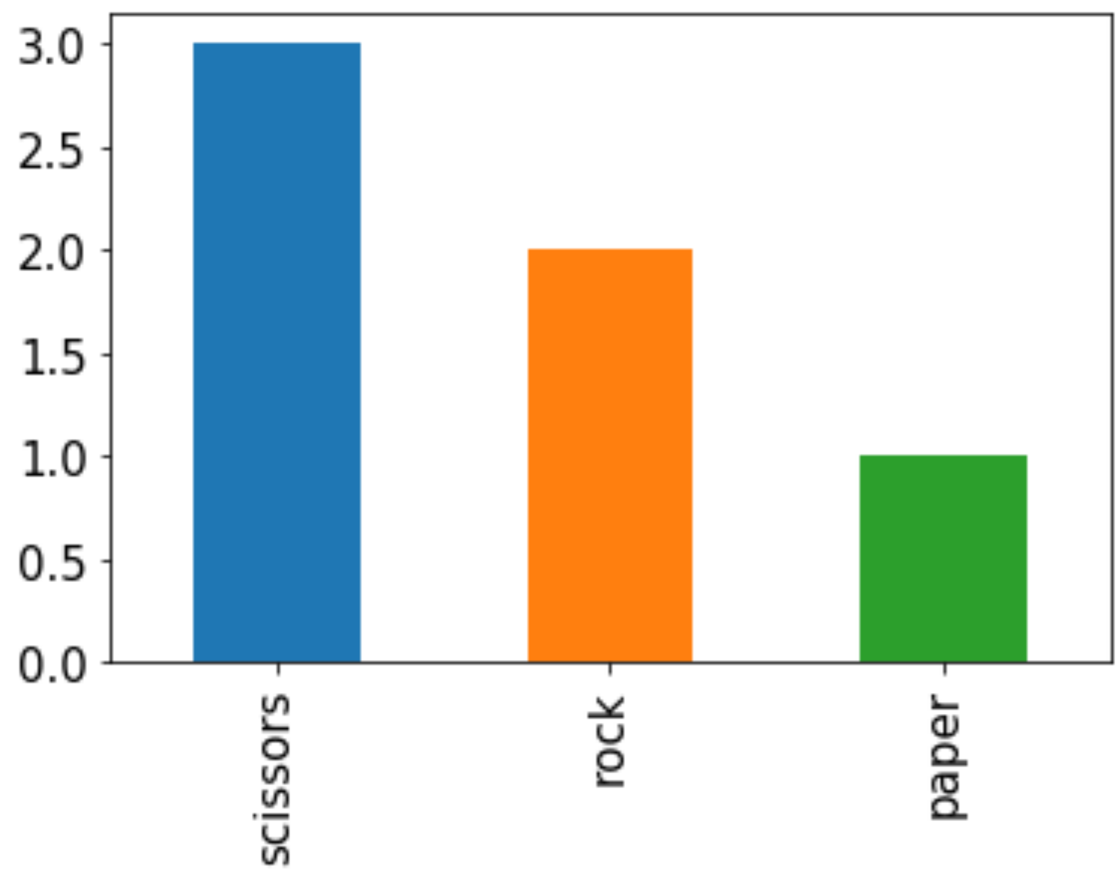
statistical significance: an intuitive approach

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()
```

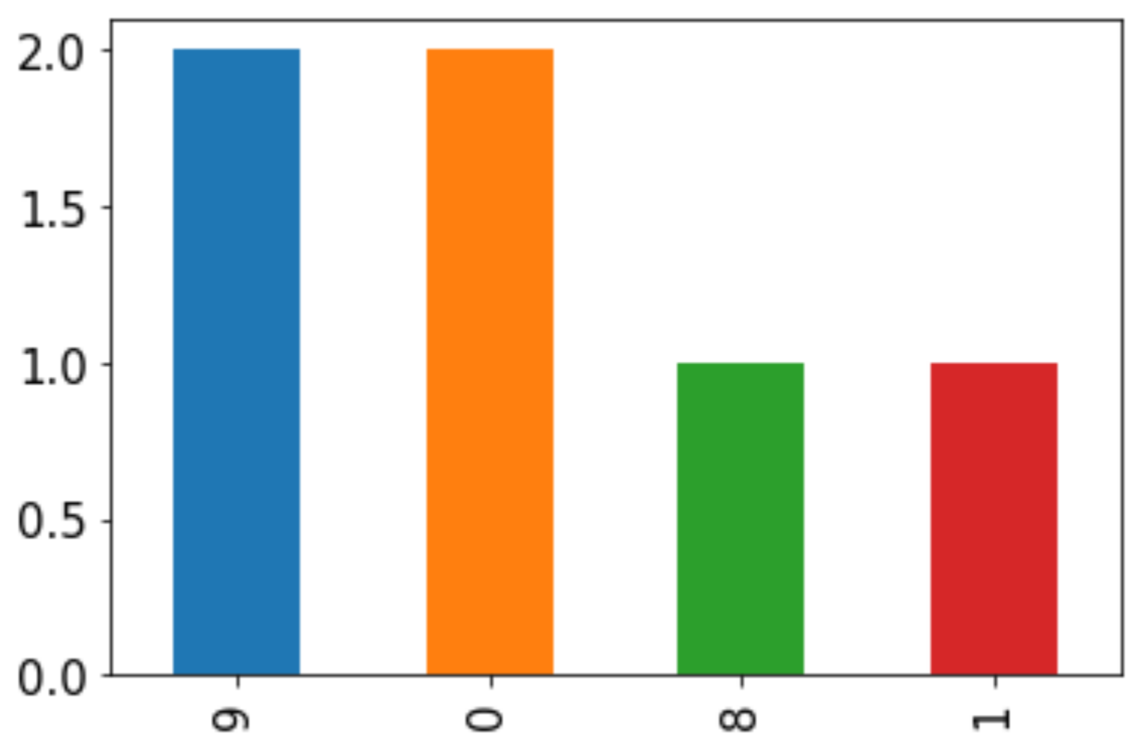


Frequencies across numbers

bars are a bad way to view frequencies across numbers

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

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



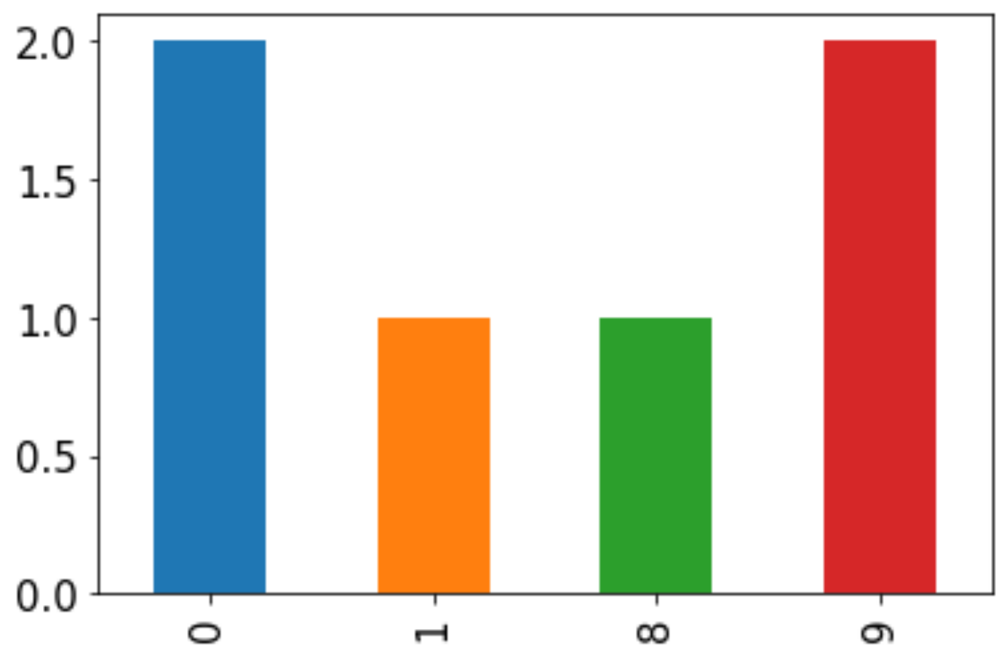
numbers not ordered

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()
```



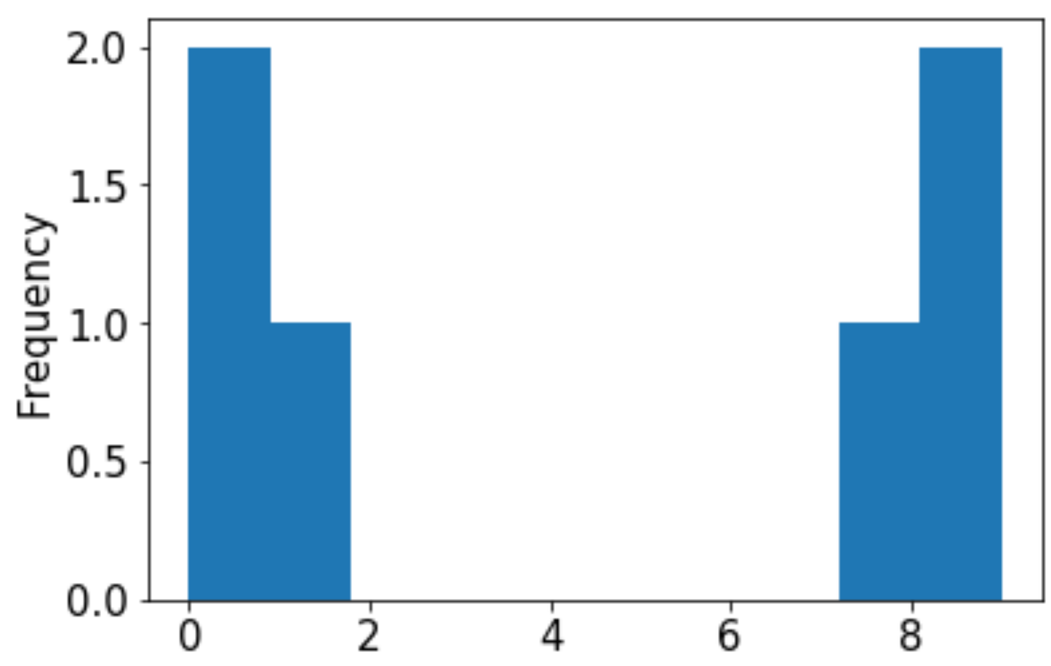
gap between 1 and 8 not obvious

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()
```



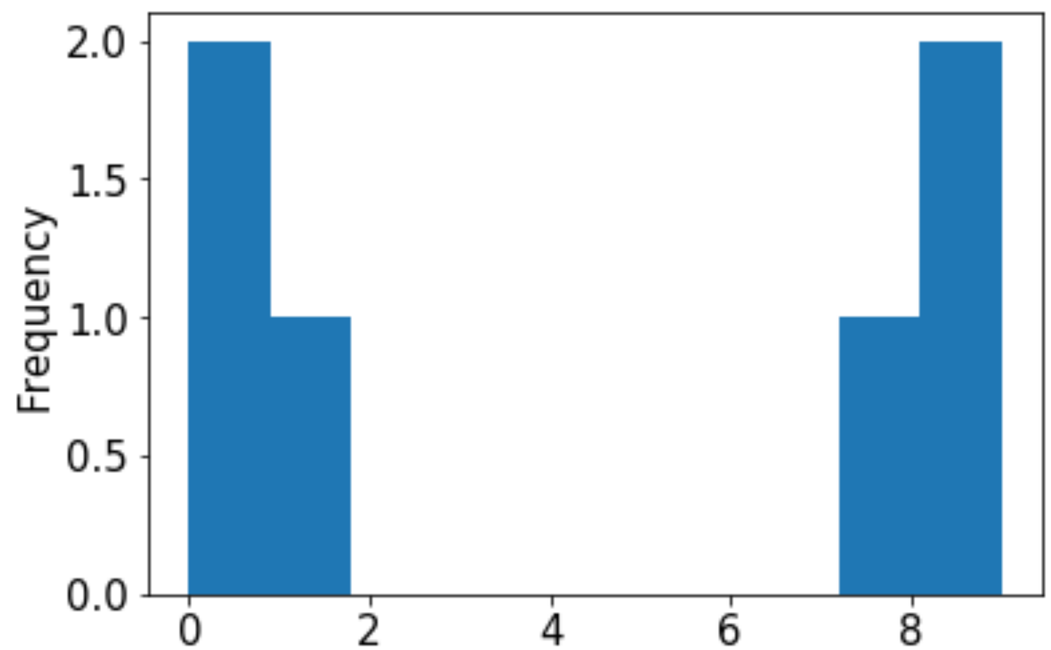
Frequencies across numbers

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

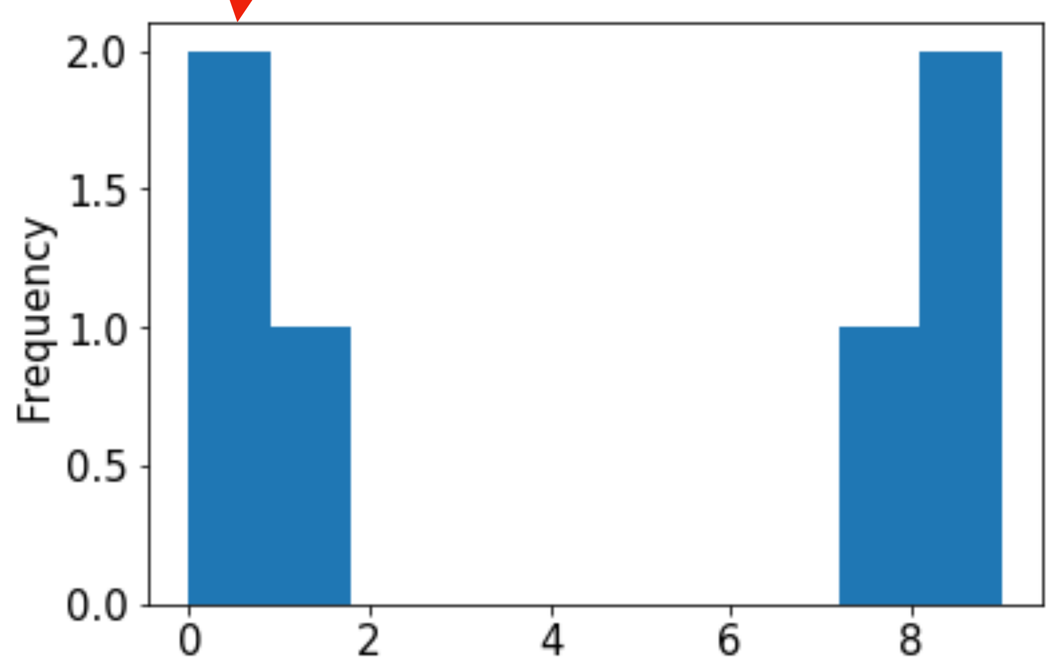
histograms are a good way to view frequencies across numbers

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

~~s.value_counts().sort_index().plot.bar()~~

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

both 0 and 0.1



a histogram "bins" nearby numbers to create discrete bars

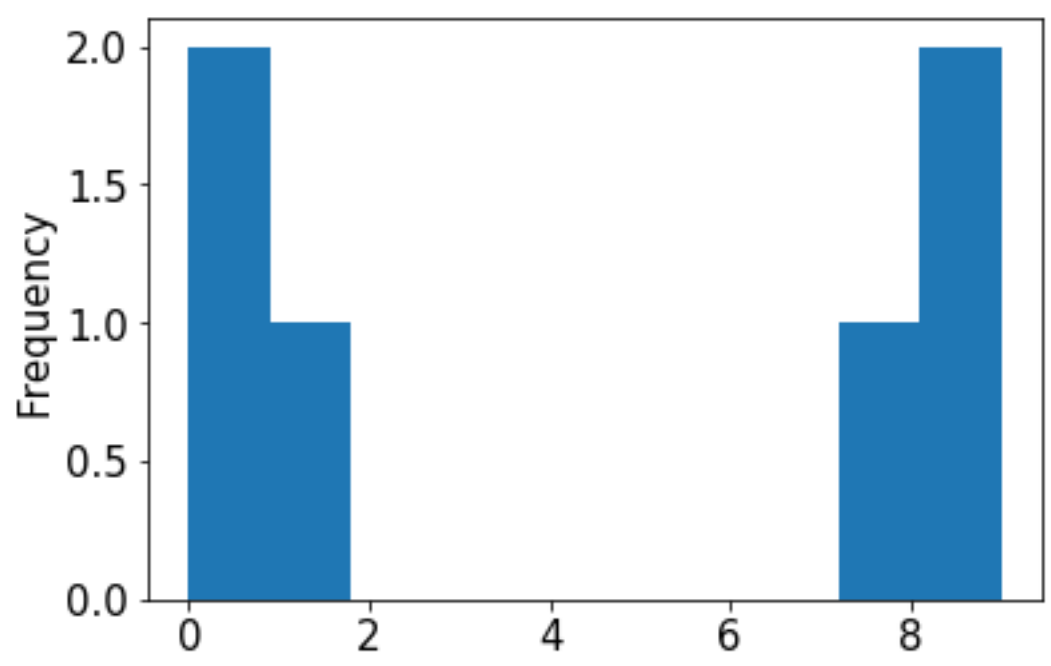
Frequencies across numbers

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

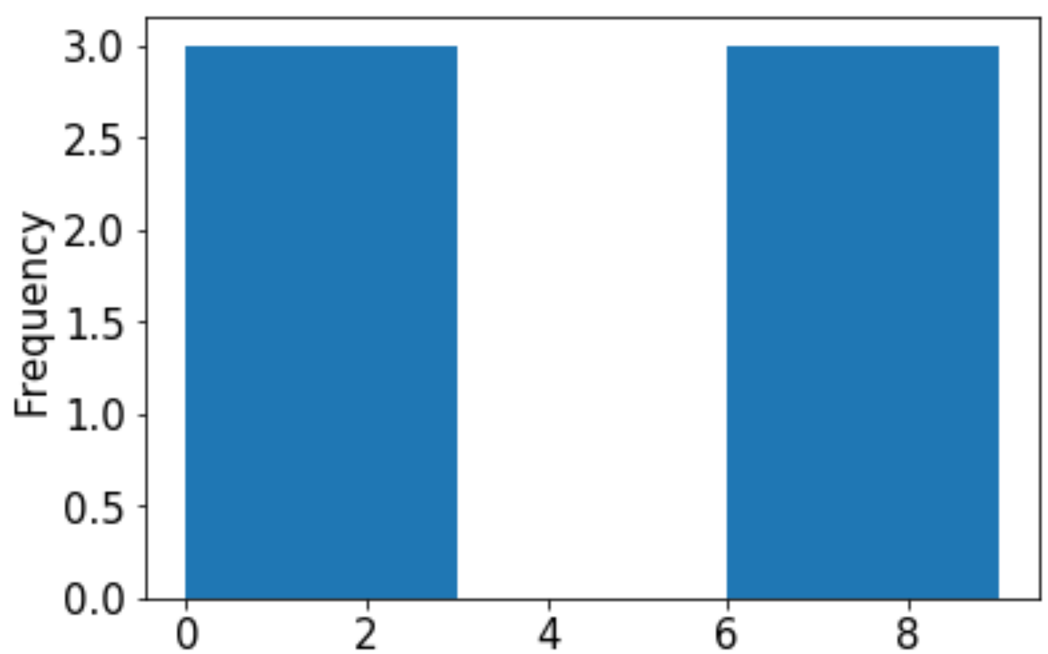
Frequencies across numbers

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

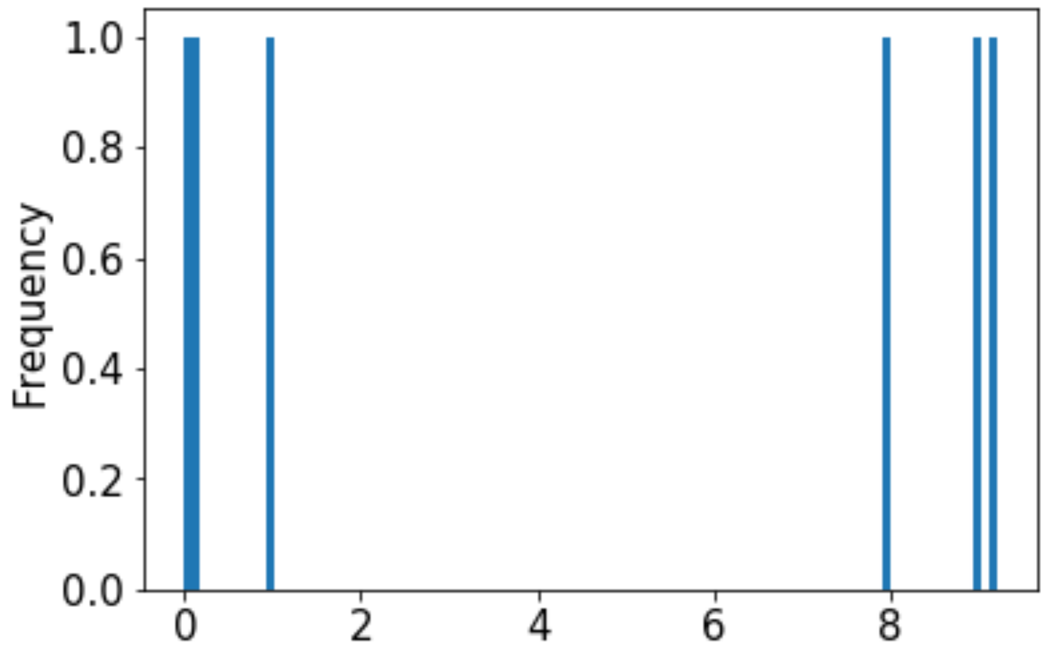
Frequencies across numbers

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)
```



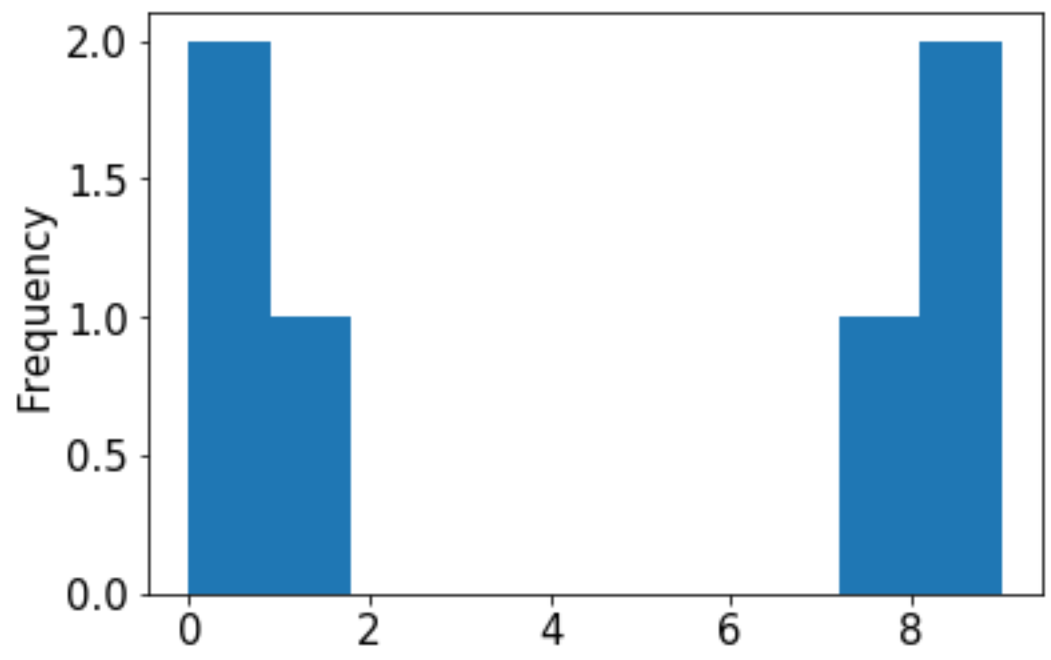
too many bins provides too much detail (equally bad)

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

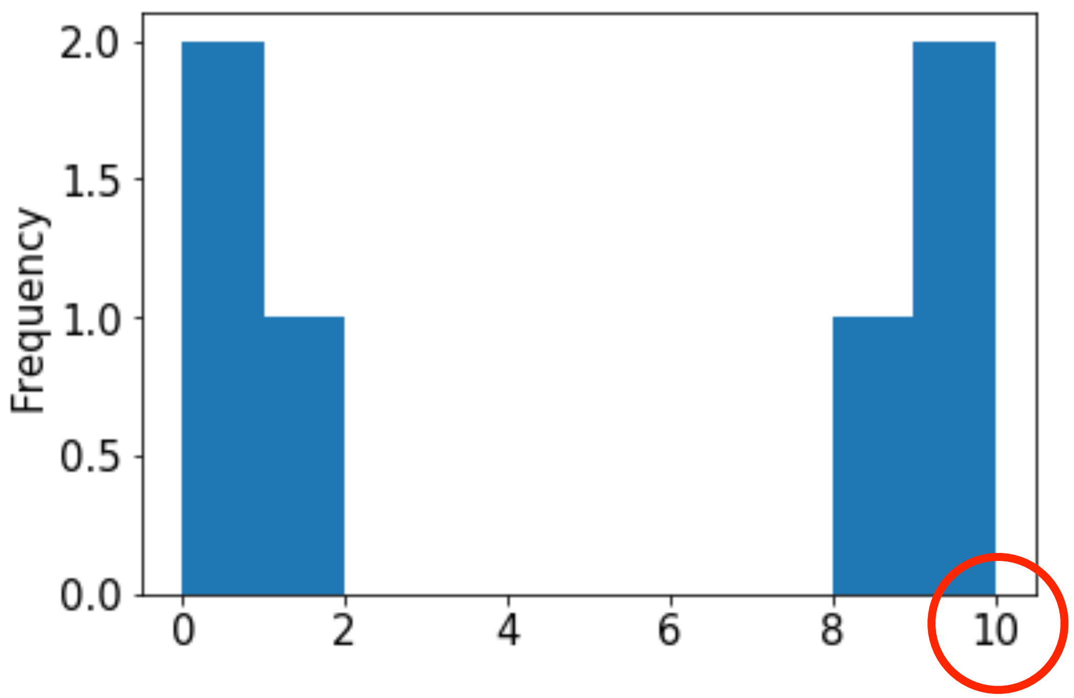
Frequencies across numbers

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])
```



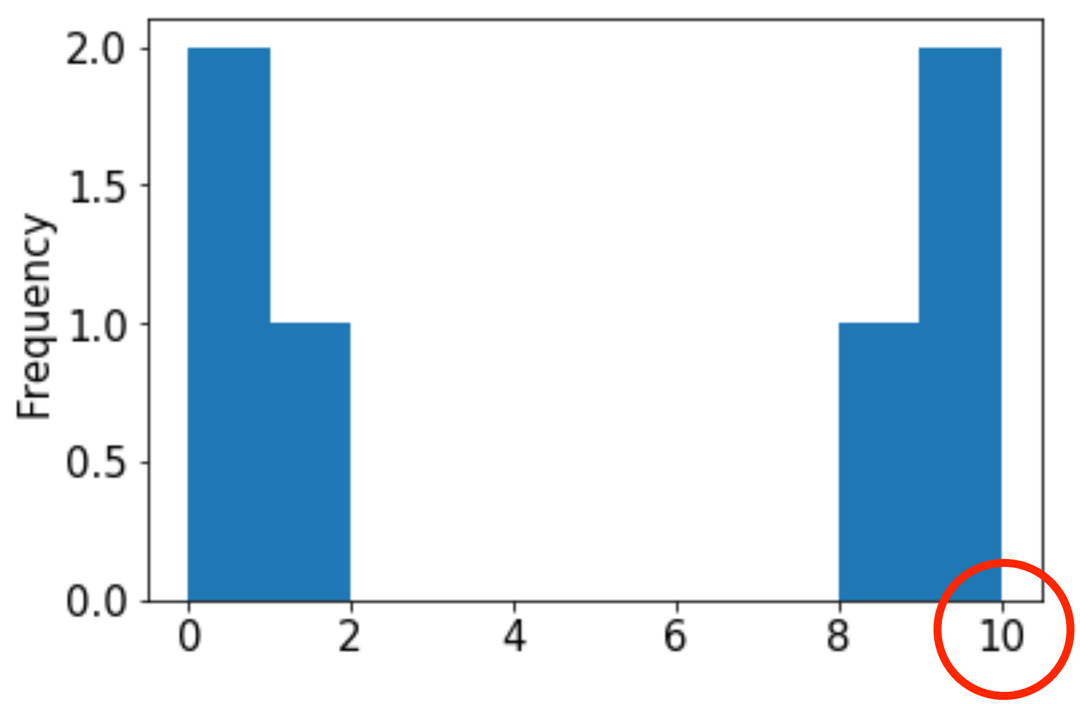
we can override the defaults

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

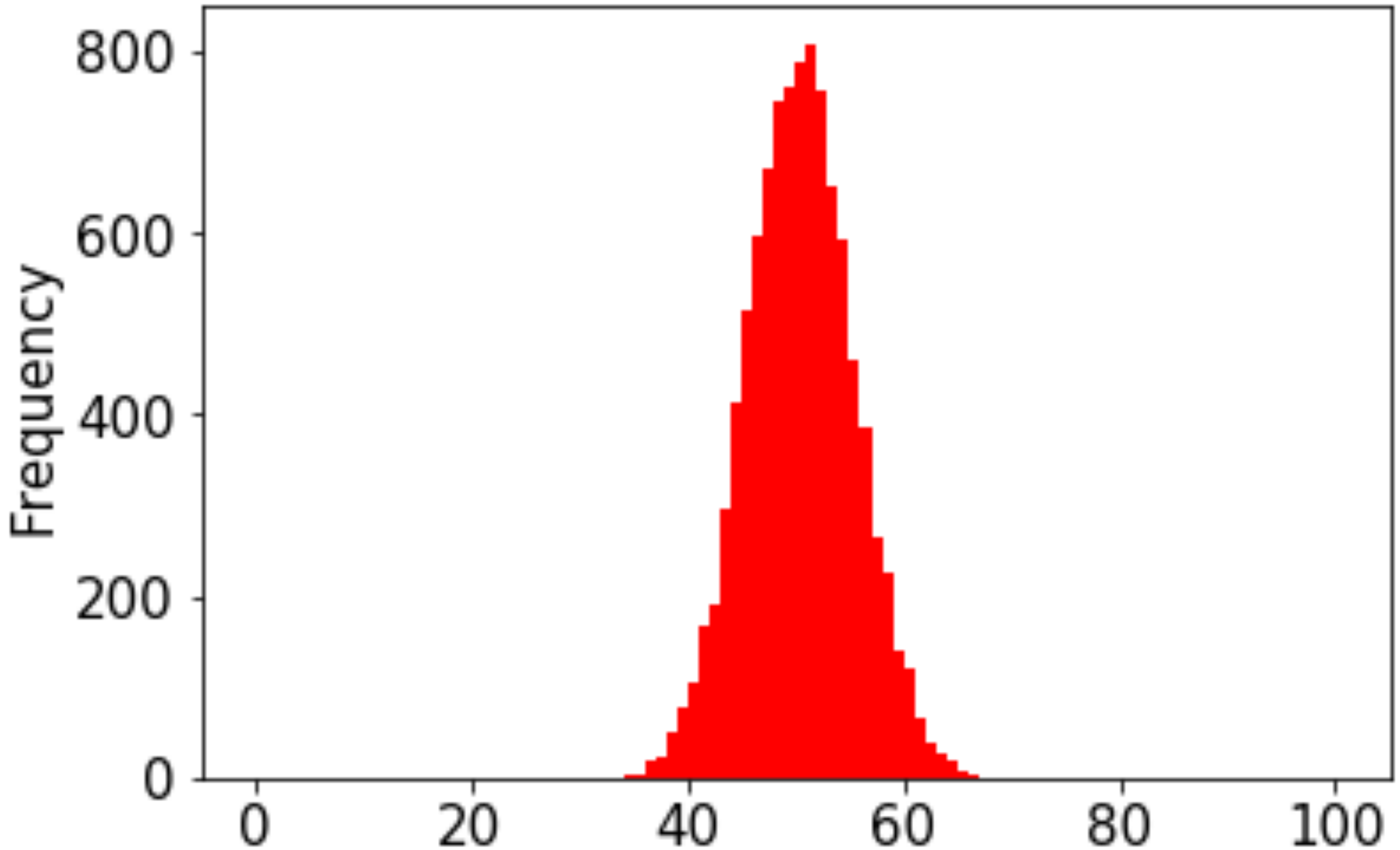
6 heads

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

number of samples

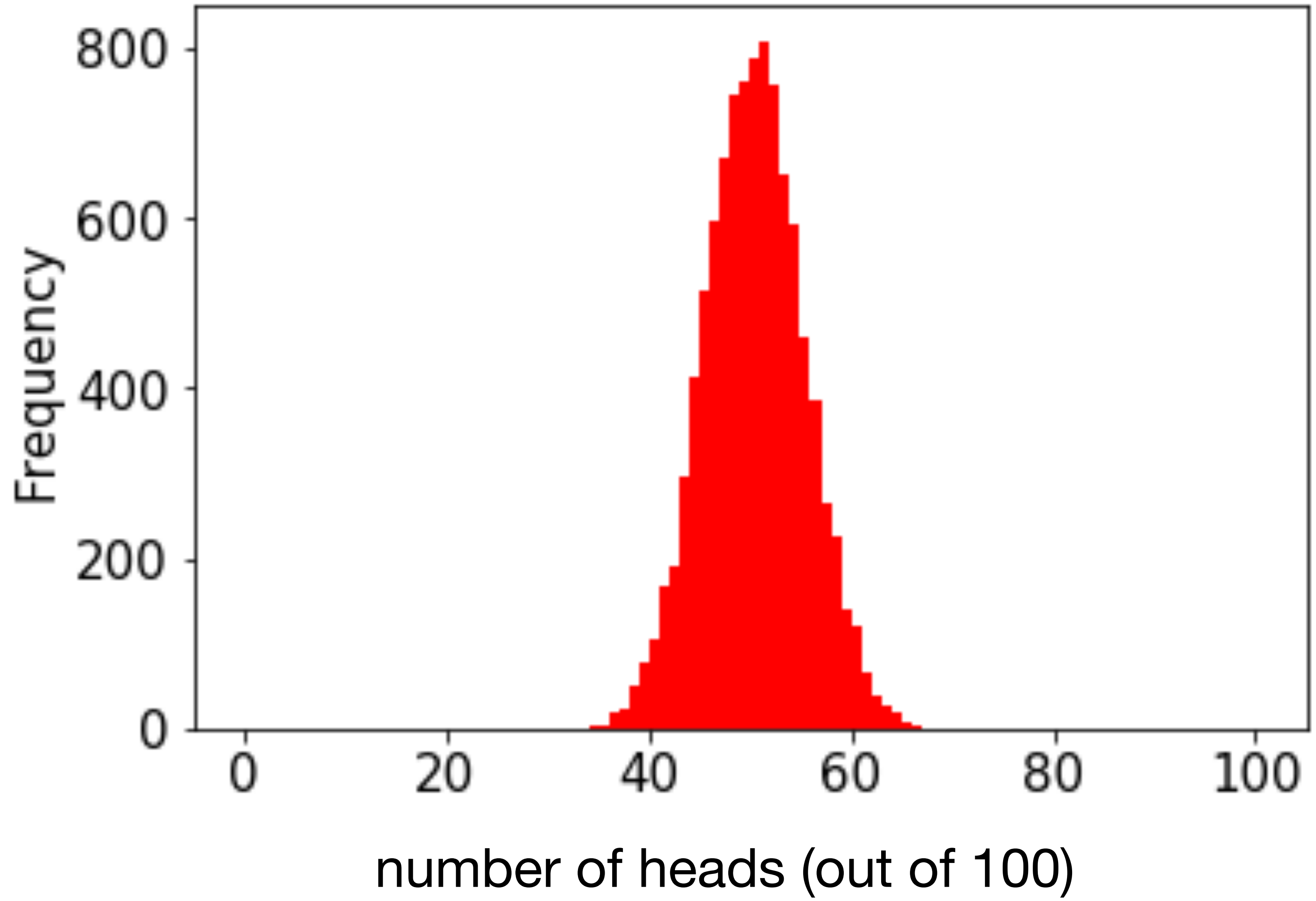
sample size

Demo 2: result



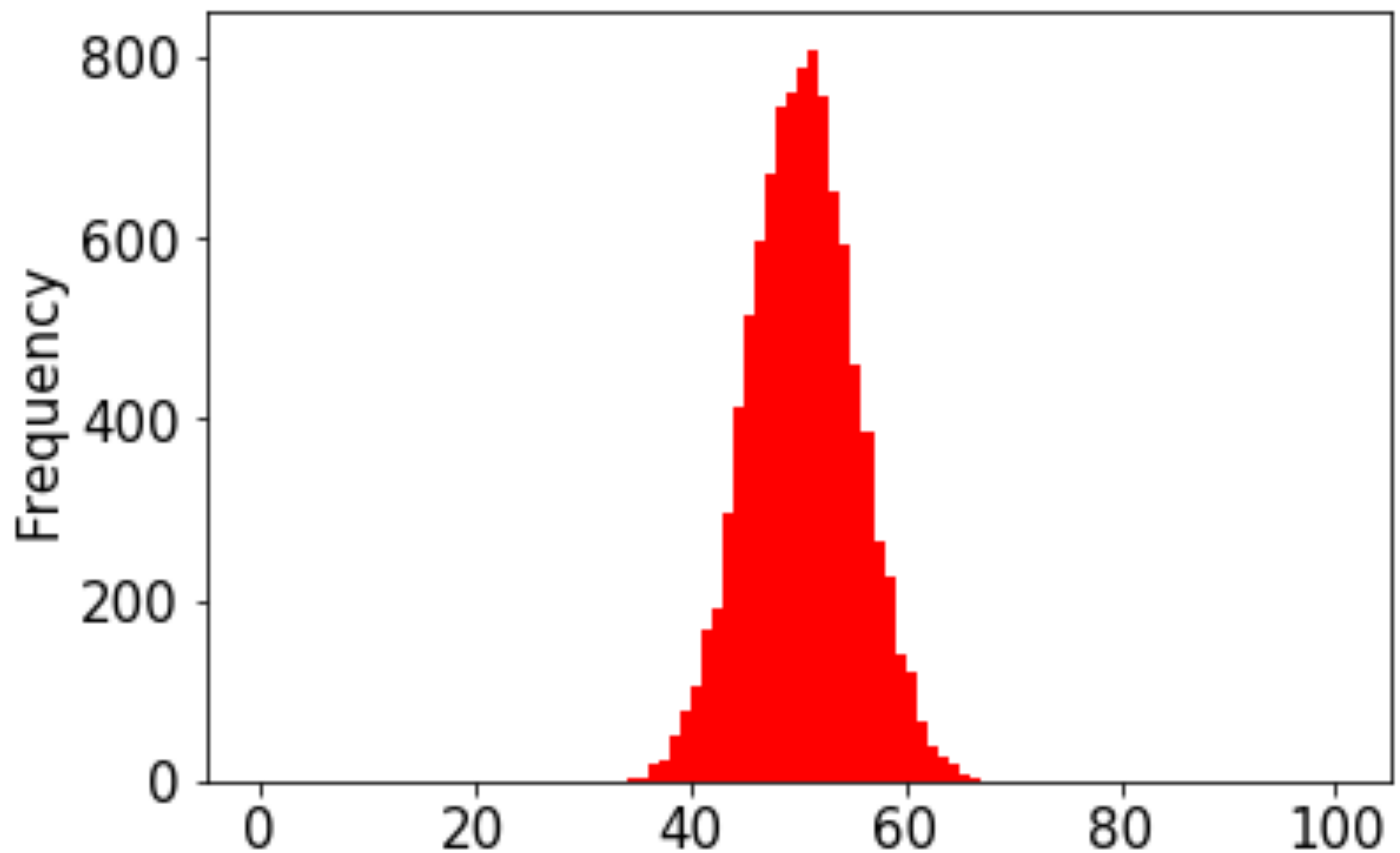
number of heads (out of 100)

Demo 2: result



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

Demo 2: result



number of heads (out of 100)

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

Outline

choice()

pseudorandom: debugging/seeding

visualization: bar plots vs. histograms

normal()

statistical significance: an intuitive approach