[220 / 319] Randomness

Meena Syamkumar
Andy Kuemmel

Announcements

- Follow-up courses
  - Direct follow up course: CS 320
  - Computer Sciences: CS 200, 300, 400, 537, 564, 640
- Office Hours
  - Last day of TA / PM office hours: Wednesday Dec 15th.
  - Additional instructor office hours

Final exam

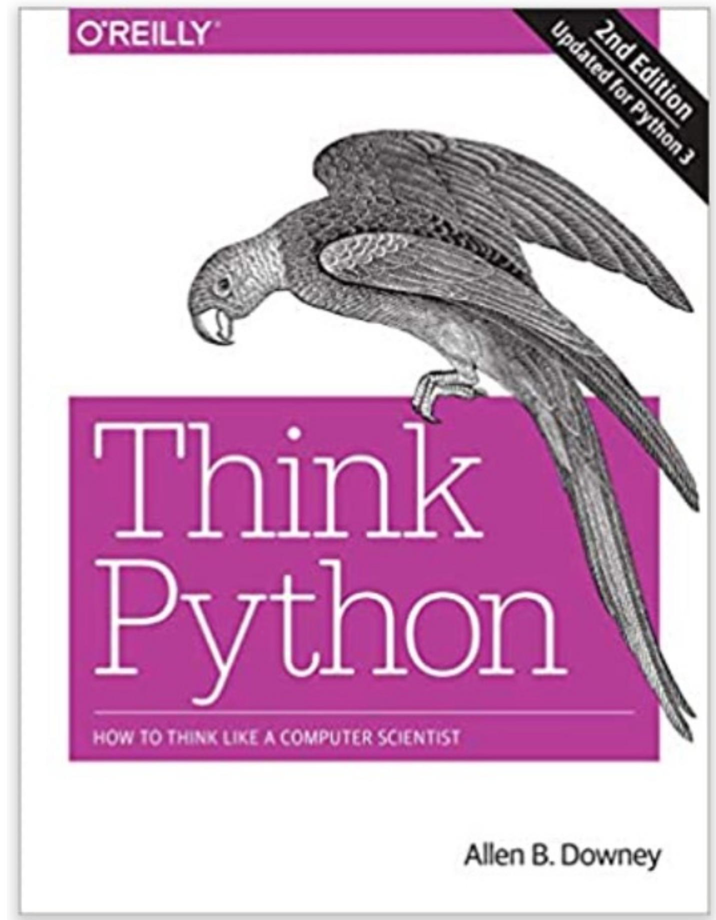
- Recommended prep
  - review past exam question papers
  - make sure you understand all the worksheet problems
  - review the readings, slides, lecture demo code
  - review everything you got wrong on the midterms
  - prepare a note sheet
- Live review session on Wednesday Dec 15th
  - All are welcome to attend

Course evaluations

- We value student feedback greatly
- Please bring a smile to your instructors' face by spending a few minutes to fill out evals ☺
- Login to https://aefis.wisc.edu/
- Find the CS220 / CS319 lecture and please provide feedback

Recommended reading

Fluent Python: Clear, Concise, and Effective Programming
by Luciano Ramalho



Think Python: How to Think Like a Computer Scientist
by Allen B. Downey

Recommended reading

Data analysis:

- Data Action: Using Data for Public Good by Sarah Williams

SQL:

- Learning SQL: Generate, Manipulate, and Retrieve Data by Alan Beaulieu
- SQL Cookbook by Anthony Molinaro

Visualization:

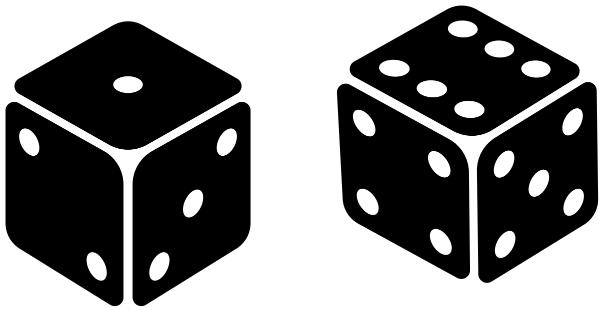
- The Visual Display of Quantitative Information by Edward R. Tufte

Statistics:

- Thinking, Fast and Slow by Daniel Kahneman
- The Signal and the Noise by Nate Silver
- Statistics Done Wrong by Alex Reinhart

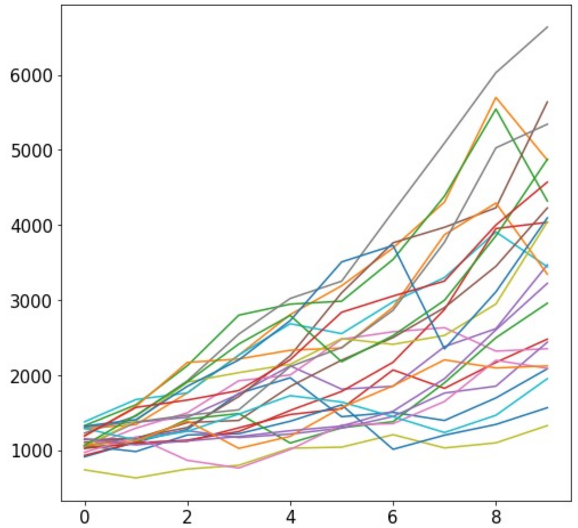
Why Randomize?

Games



Security

Simulation



our focus

Outline

choice()

bugs and seeding

significance

histograms

normal()

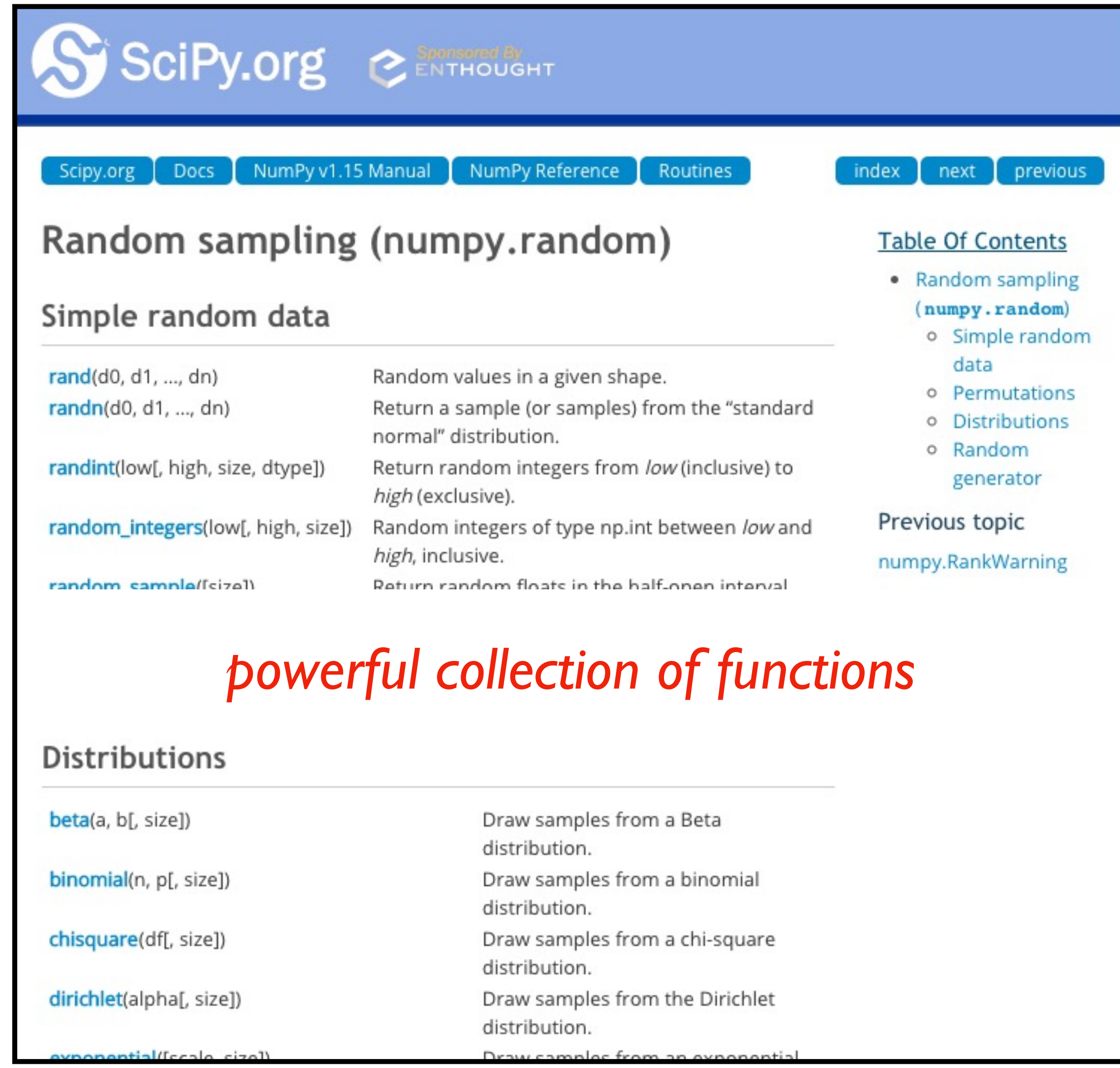
New Functions Today

`numpy.random:`

- powerful collection of functions
- **`choice`**

`Series.plot.hist:`

- similar to bar plot
- visualize spread of random results



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

https://www.securifera.com/blog/2015/09/09/mmactf-2015-rock-paper-scissors-rps/

choice

```
from numpy.random import choice

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

https://www.securifera.com/blog/2015/09/09/mmactf-2015-rock-paper-scissors-rps/

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

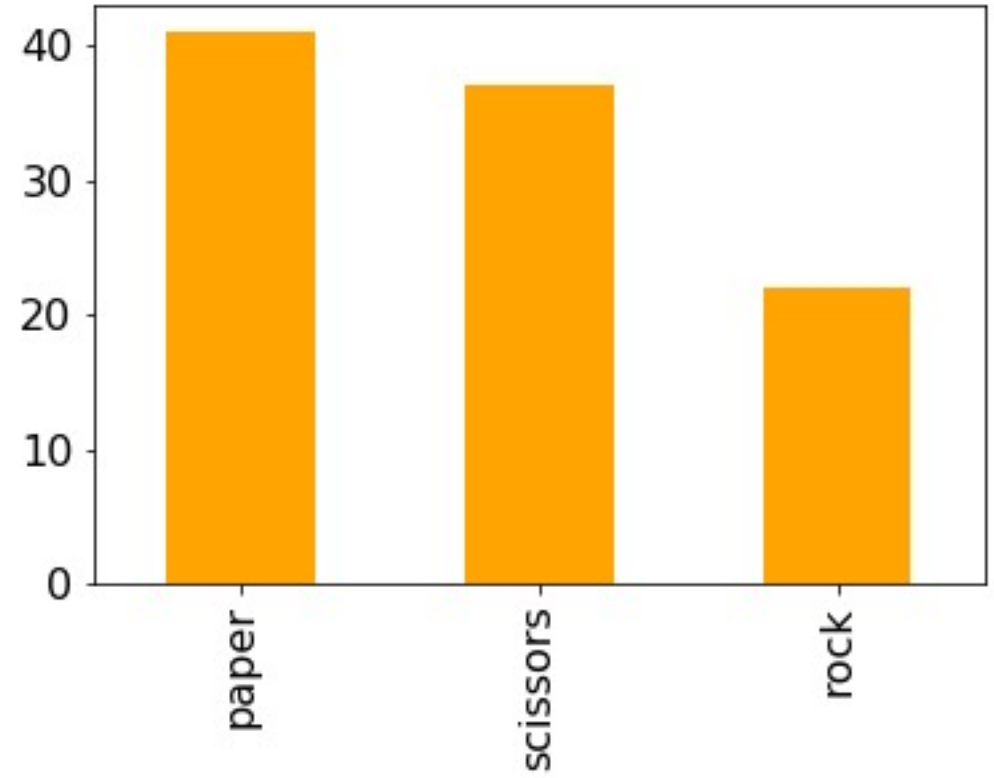
```
from numpy.random import choice

# 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?

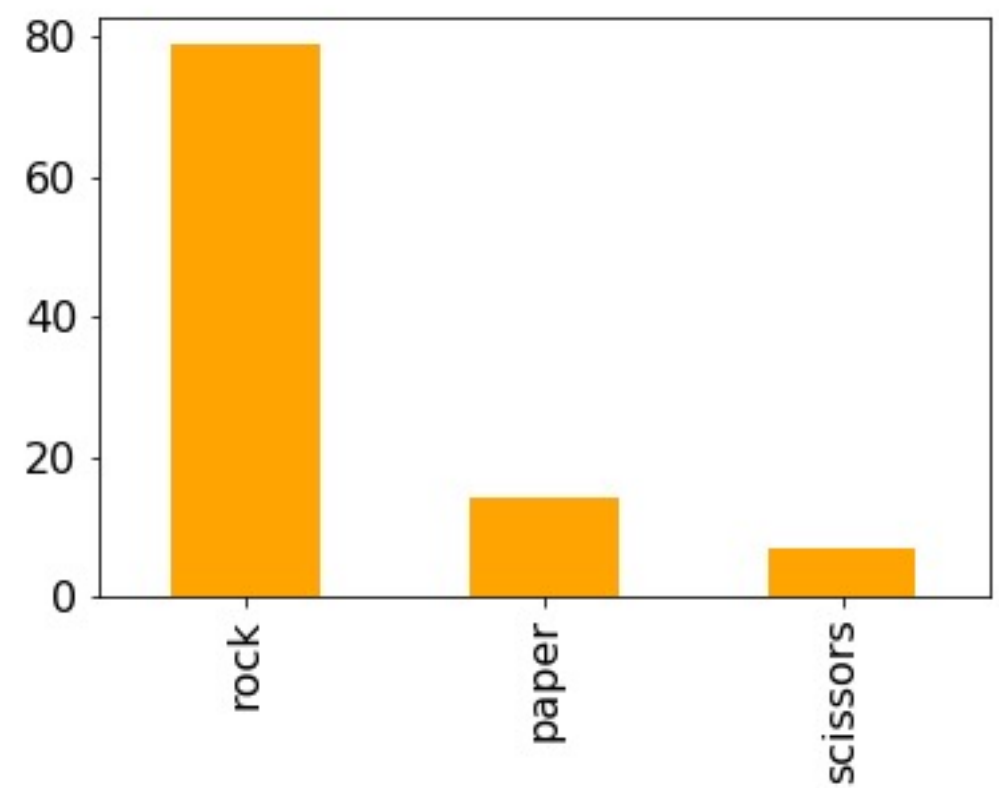


Demo: exploring bias

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

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

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])
```

same

```
# 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

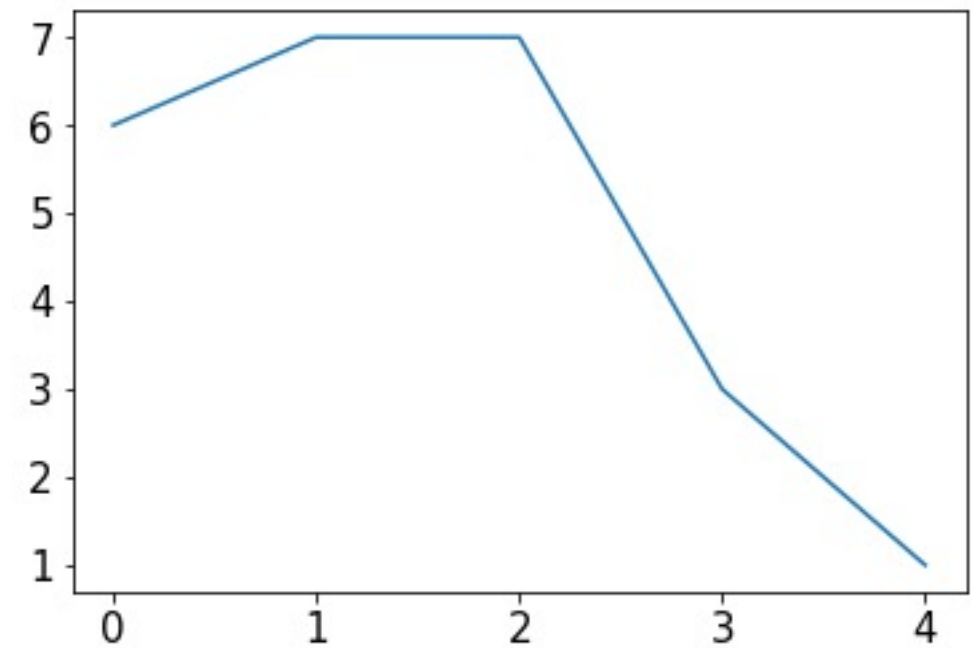
normal()

Example: change over time

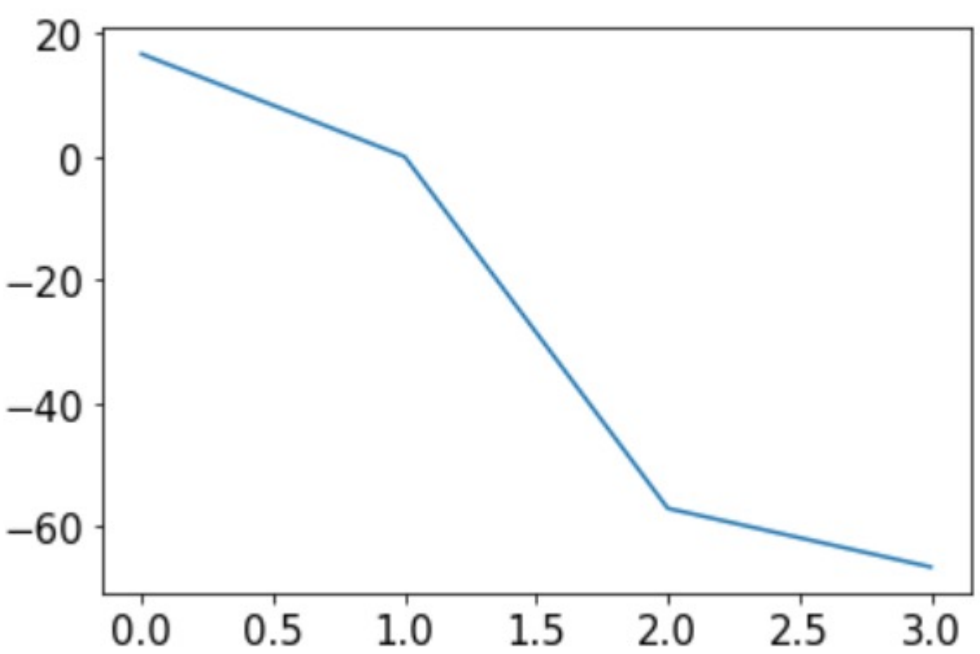
```
s = Series(choice(10, size=5))
```

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

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




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



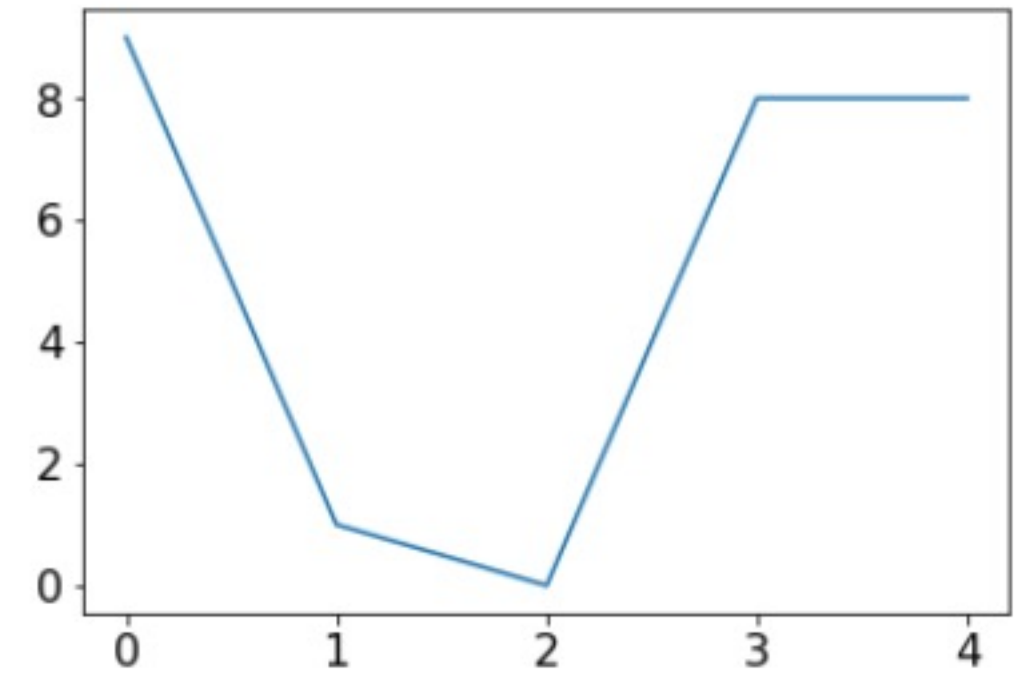
can you identify the bug in the code?

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()
```

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

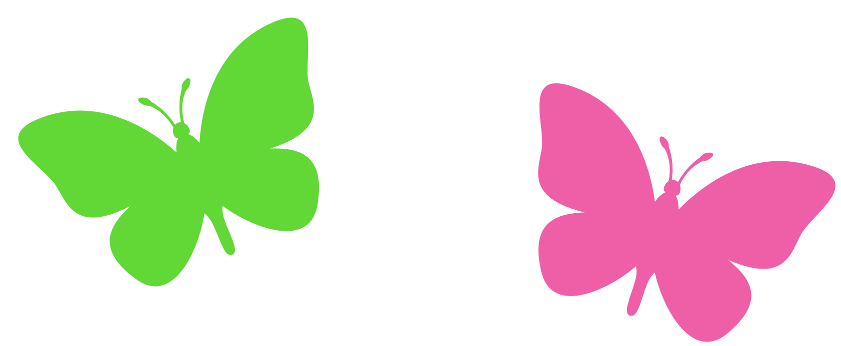
can you identify the bug in the code?

Not all bugs are equal!

scary bugs		"nice" bugs
non-deterministic system related randomness	seeding	deterministic (reproducible)
large data		small data
semantic	assert	runtime
		syntax

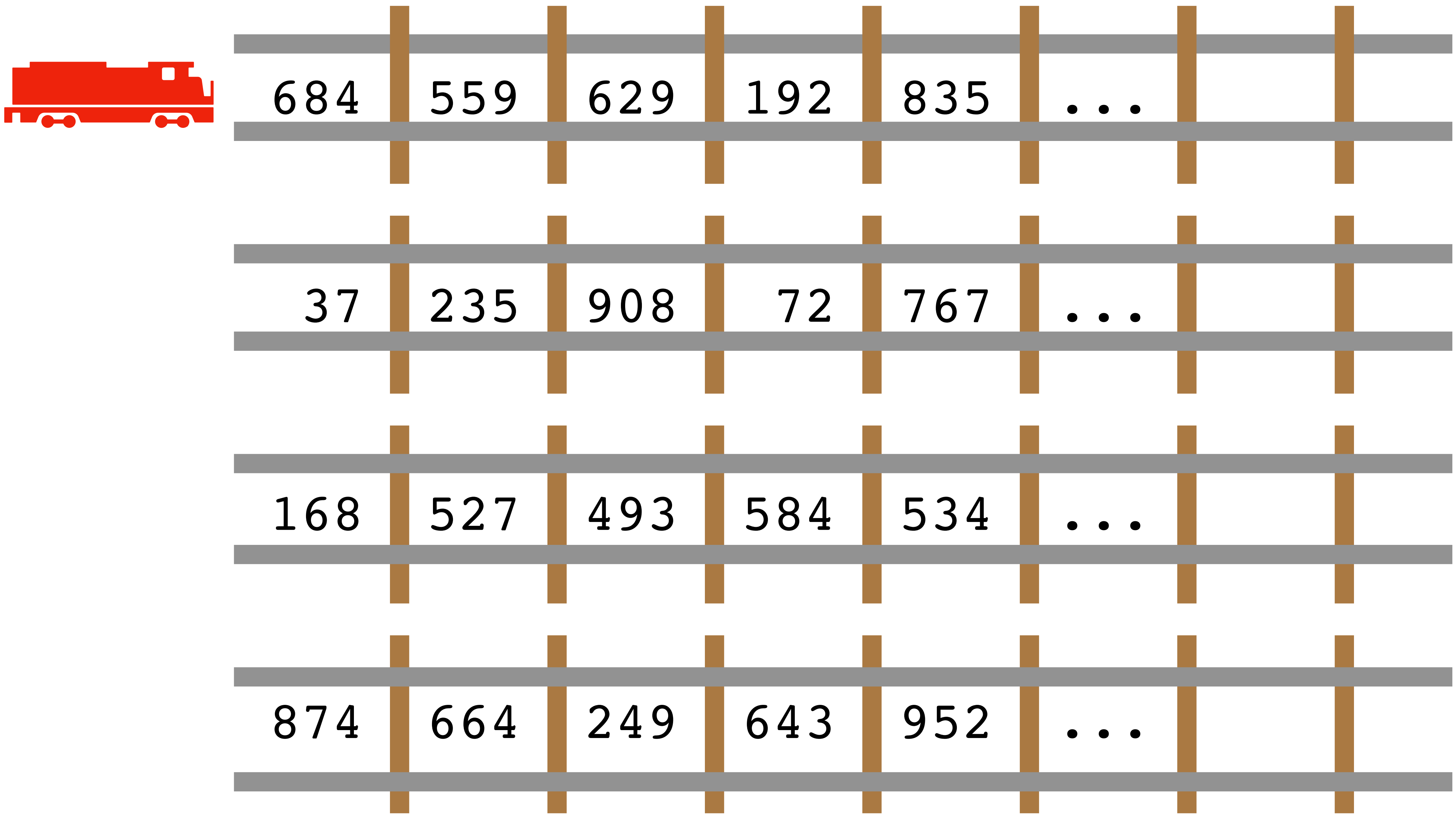
Igor Siwanowicz

https://owlcation.com/stem/5-Badass-Bugs-That-You-Should-Have-Nightmares-About



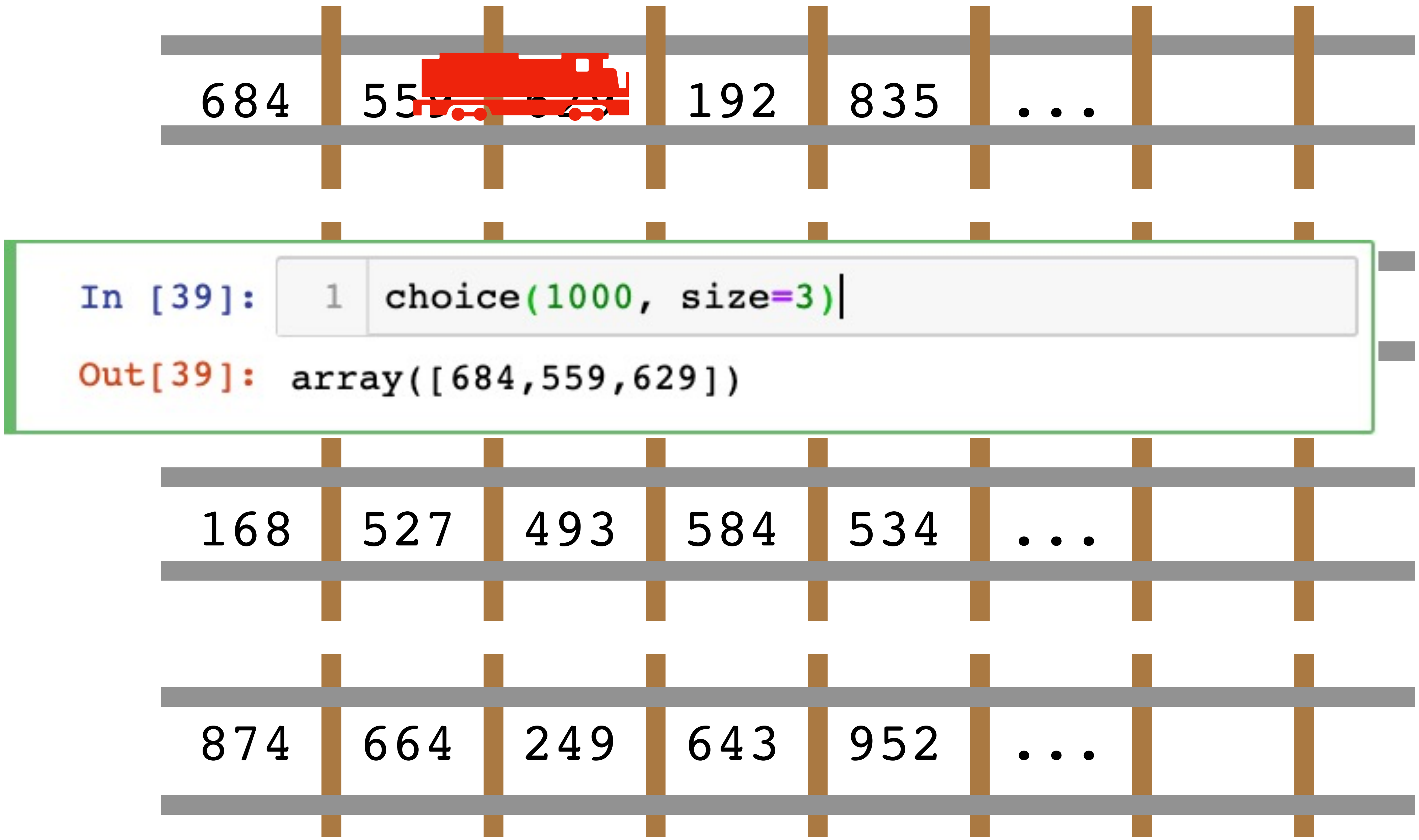
Pseudorandom Generators

"Random" generators are really just *pseudorandom*



Pseudorandom Generators

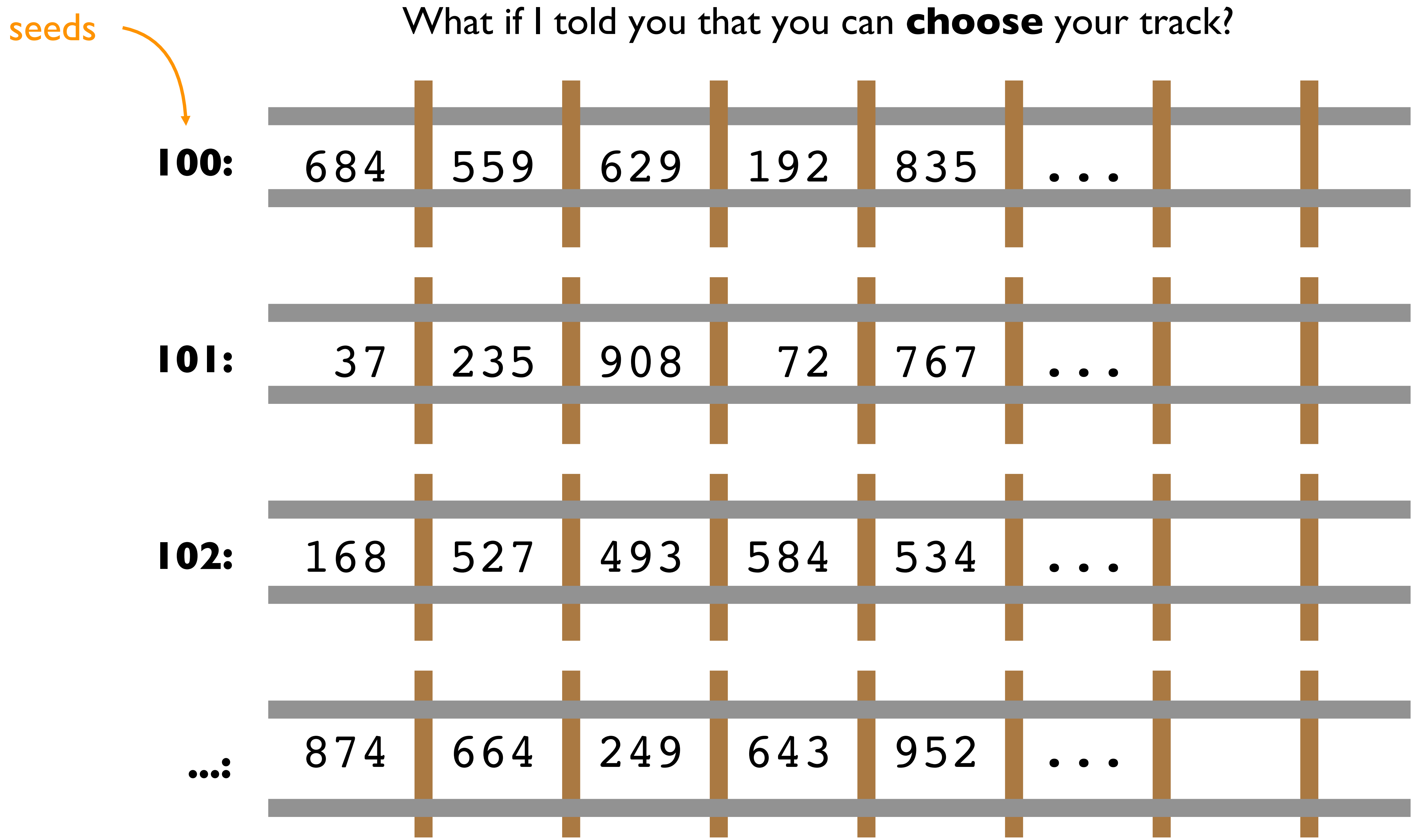
"Random" generators are really just *pseudorandom*



Pseudorandom Generators

"Random" generators are really just *pseudorandom*

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



Seeding

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

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

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

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

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

```
In [4]: np.random.seed(220)
        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]:
import time
now = int(time.time())
print("seeding with", now)
np.random.seed(now)
choice(1000, size=3)

seeding with 1556673136

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

Outline

choice()

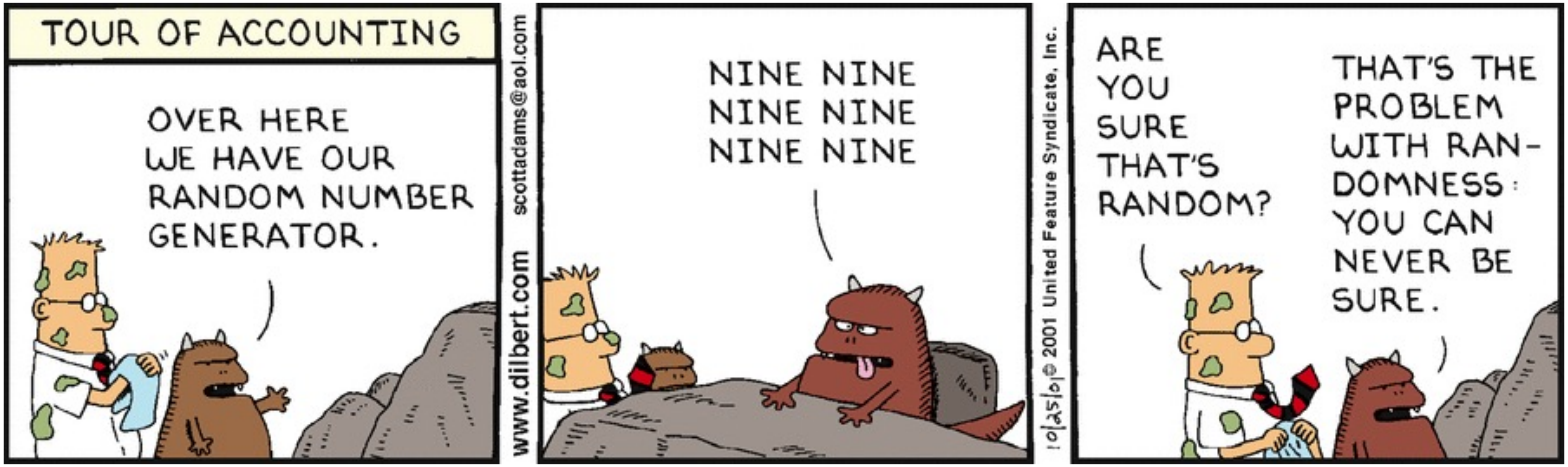
bugs and seeding

significance

histograms

normal()

In a noisy world, what is noteworthy?



https://dilbert.com/strip/2001-10-25

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.

5	**95**

Call shenanigans? Yes.

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

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?

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

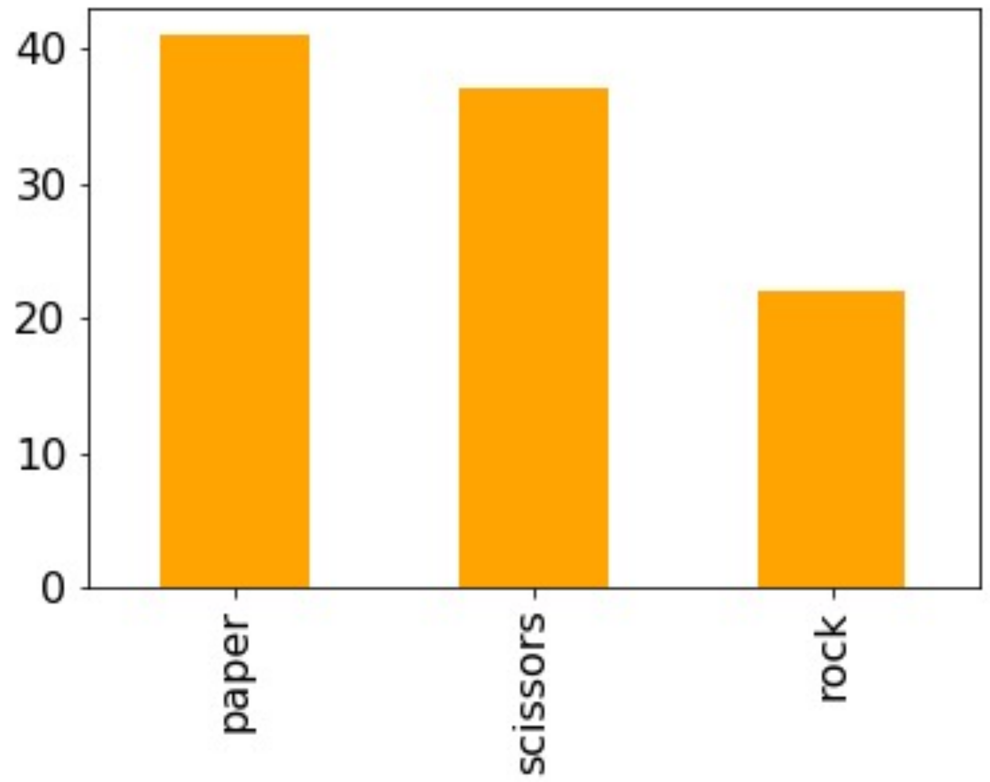
histograms

normal()

Frequencies across categories

bars are a good way to view frequencies across categories

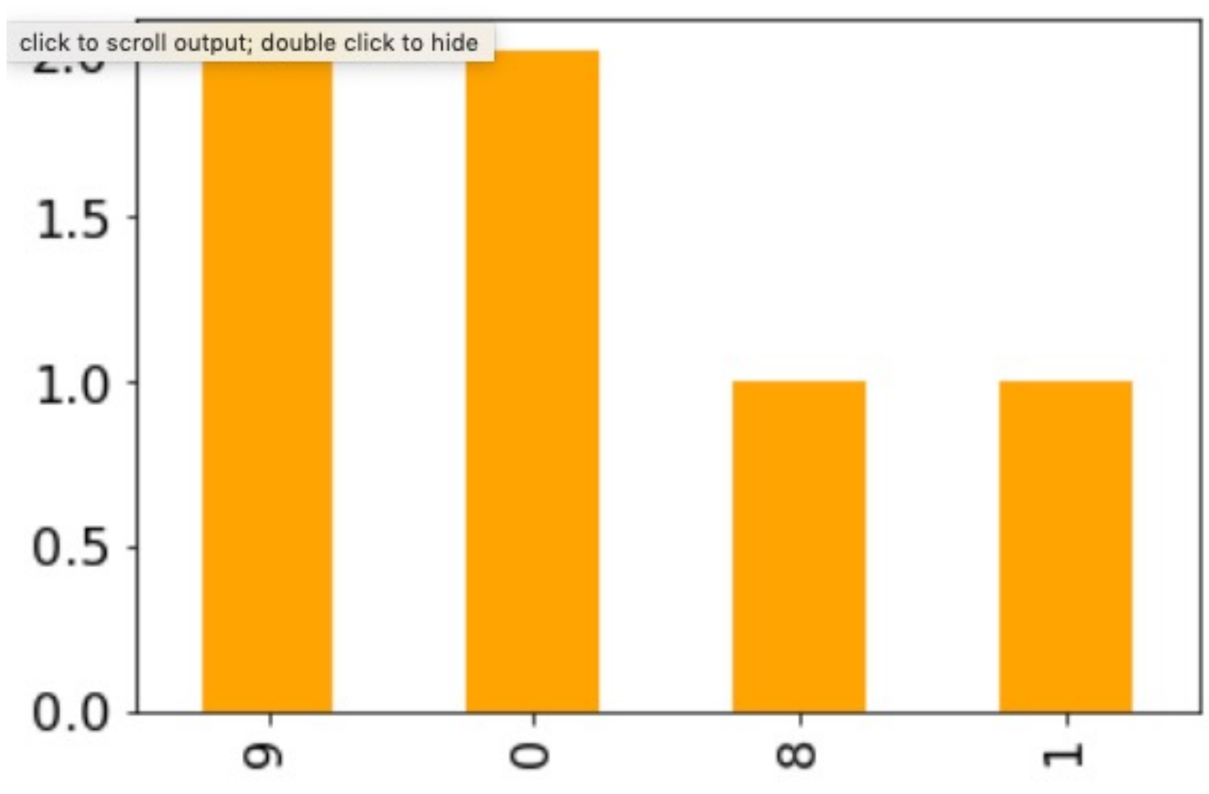
```
s = Series(["rock", "rock", "paper",
            "scissors", "scissors", "scissors"])

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")
```

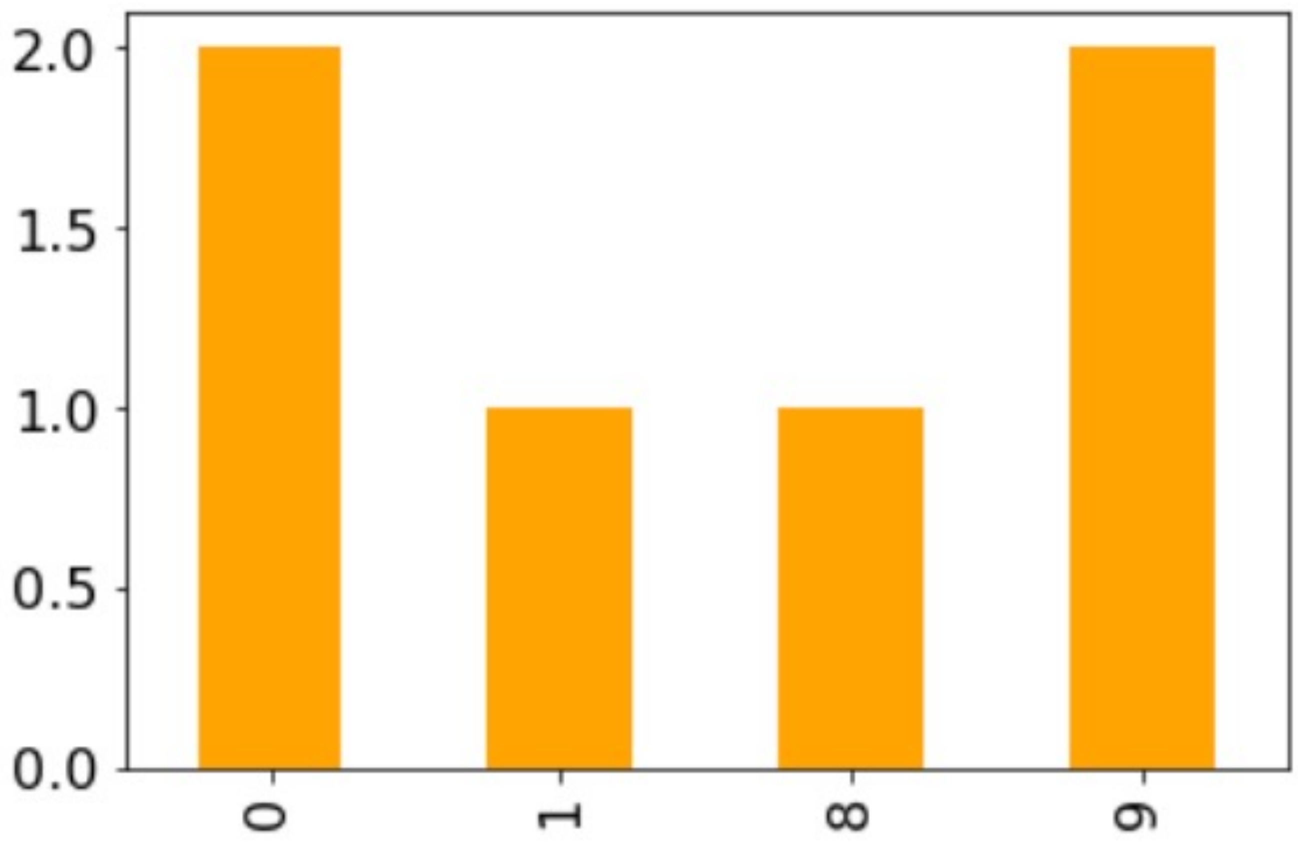


numbers not ordered

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

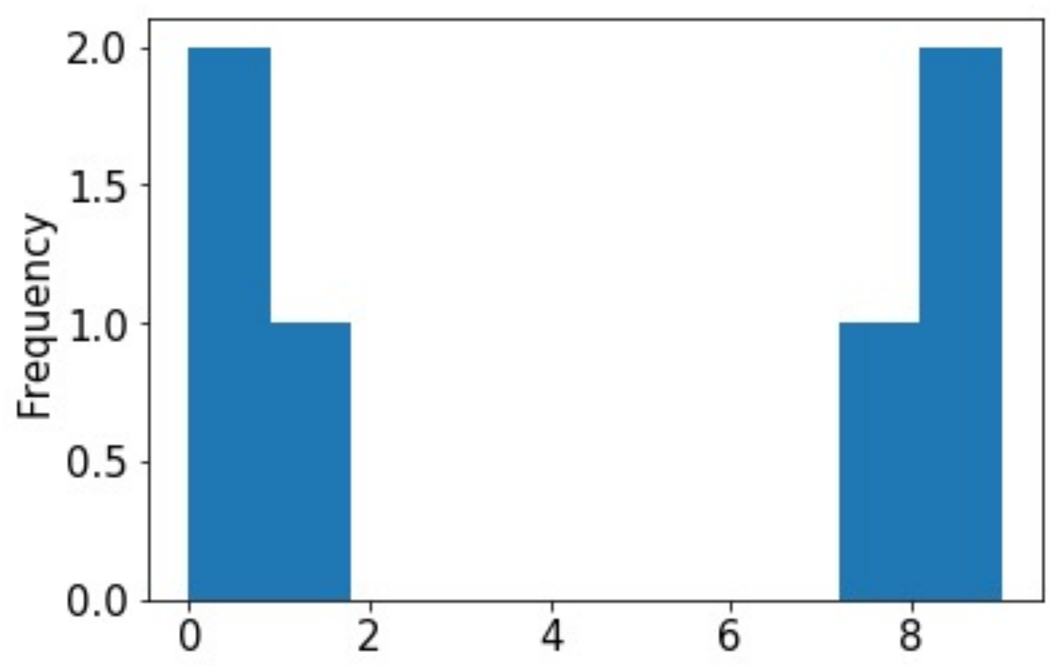
Frequencies across numbers

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

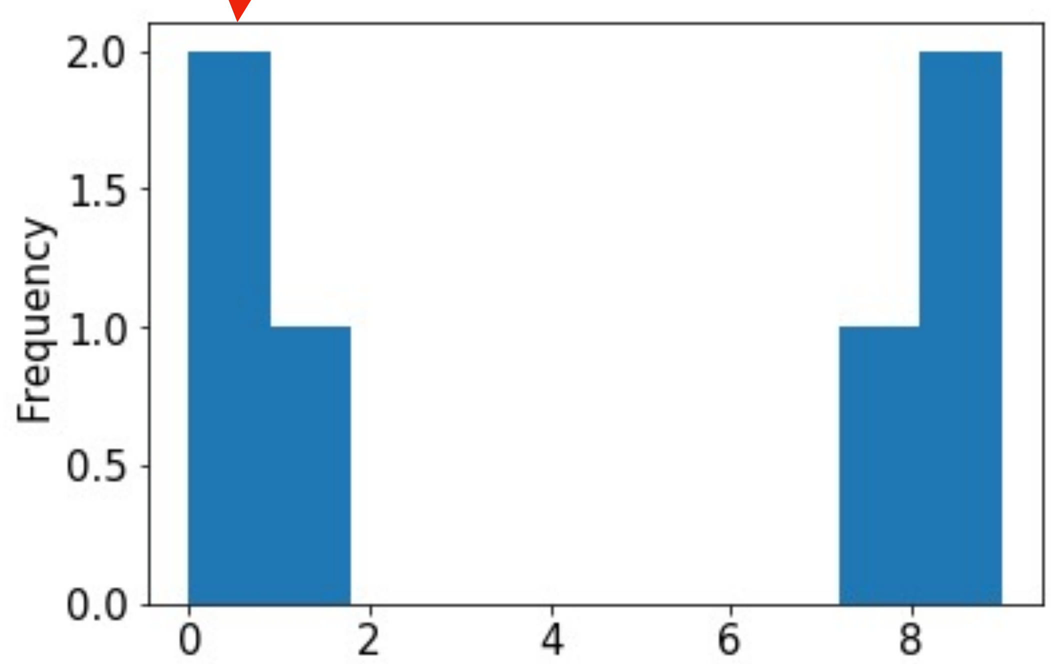
histograms are a good way to view frequencies across numbers

```
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

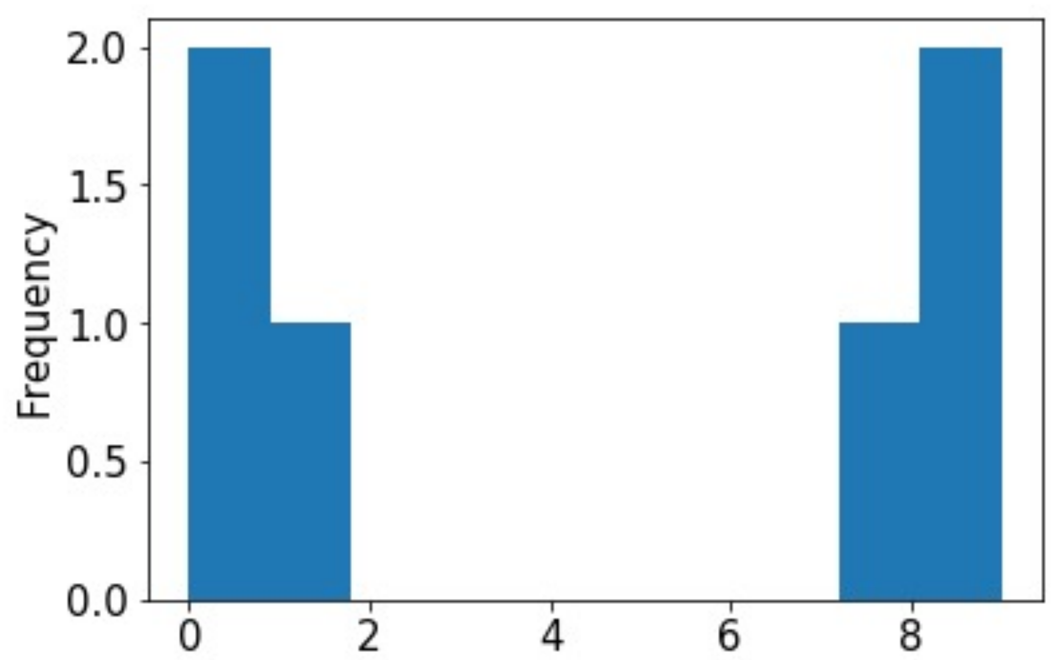


a histogram "bins" nearby numbers to create discrete bars

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)
```



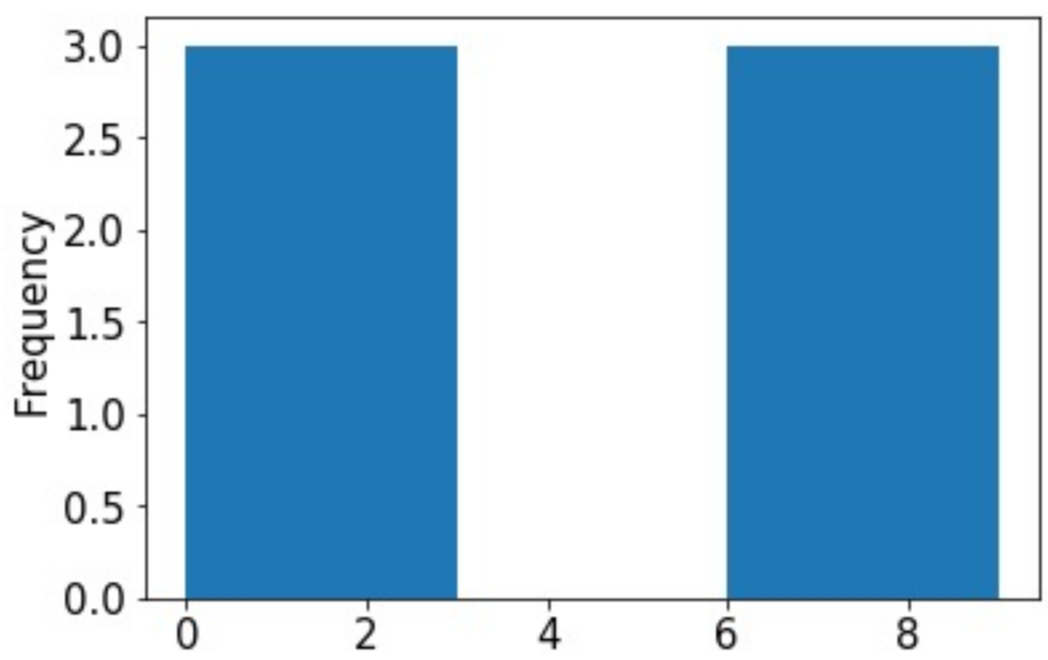
we can control the number of bins

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)
```

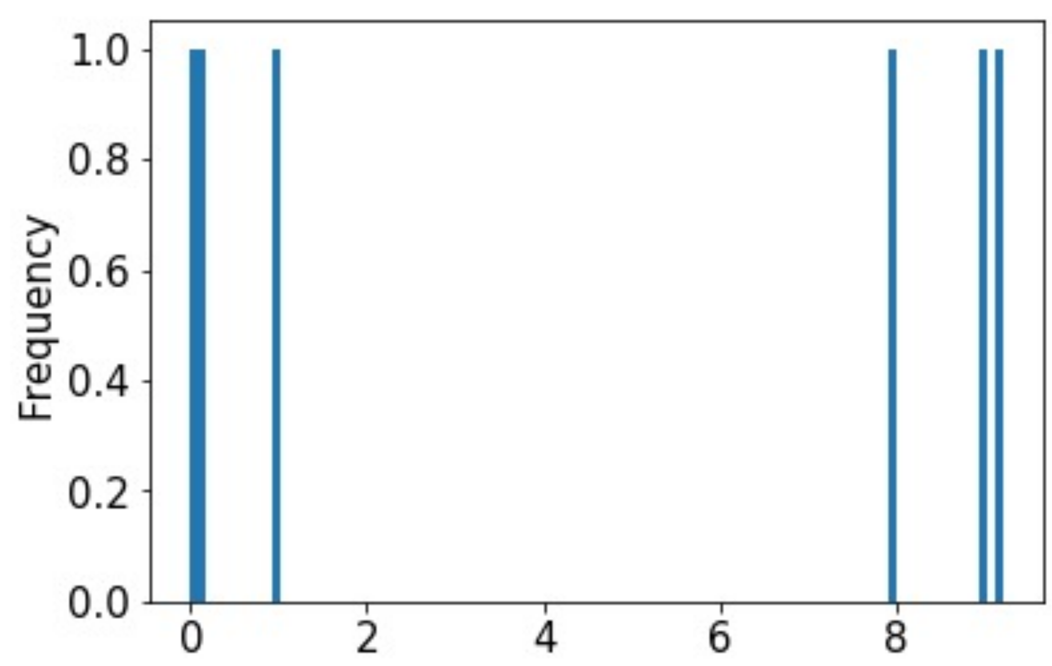


too few bins provides too little detail

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)
```

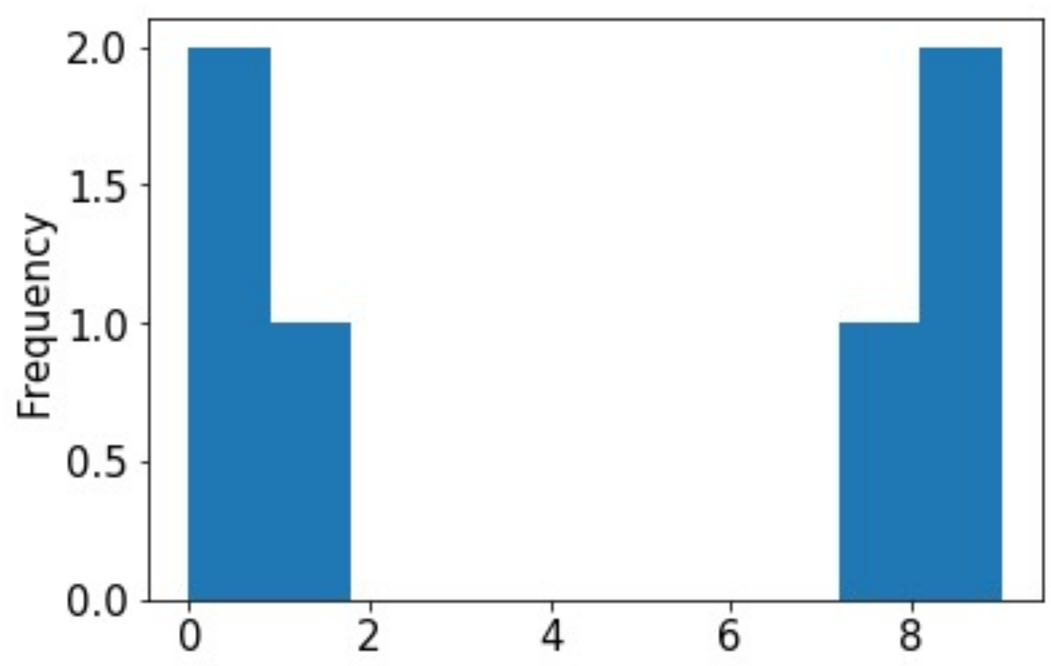


too many bins provides too much detail (equally bad)

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)
```



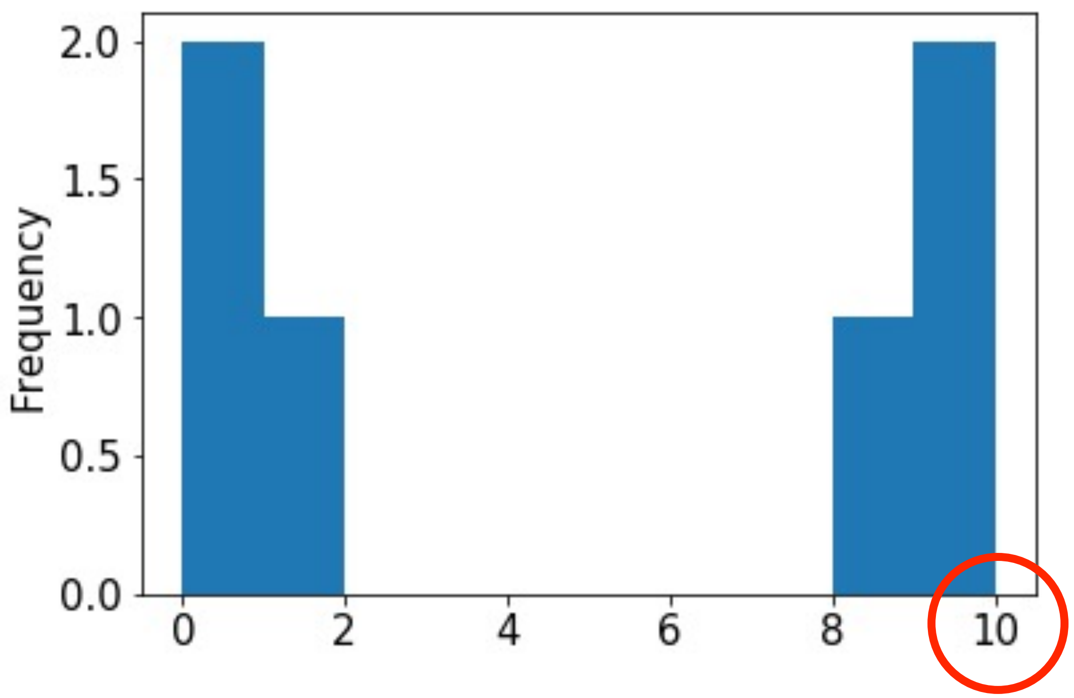
pandas chooses the default bin boundaries

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])
```

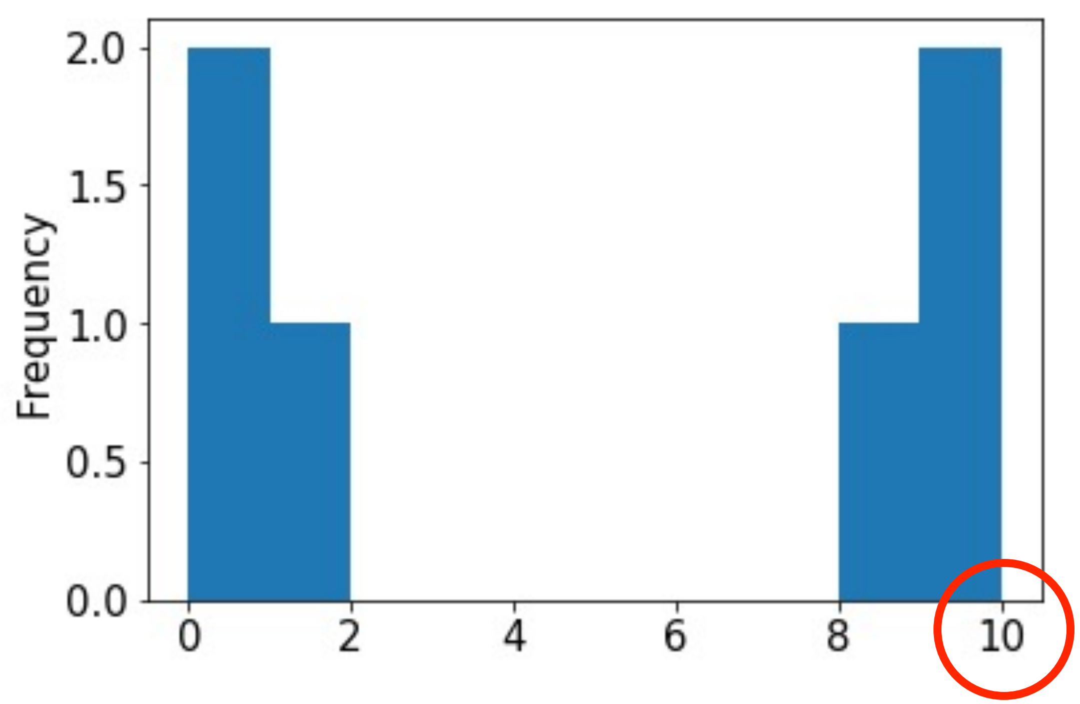


we can override the defaults

Frequencies across numbers

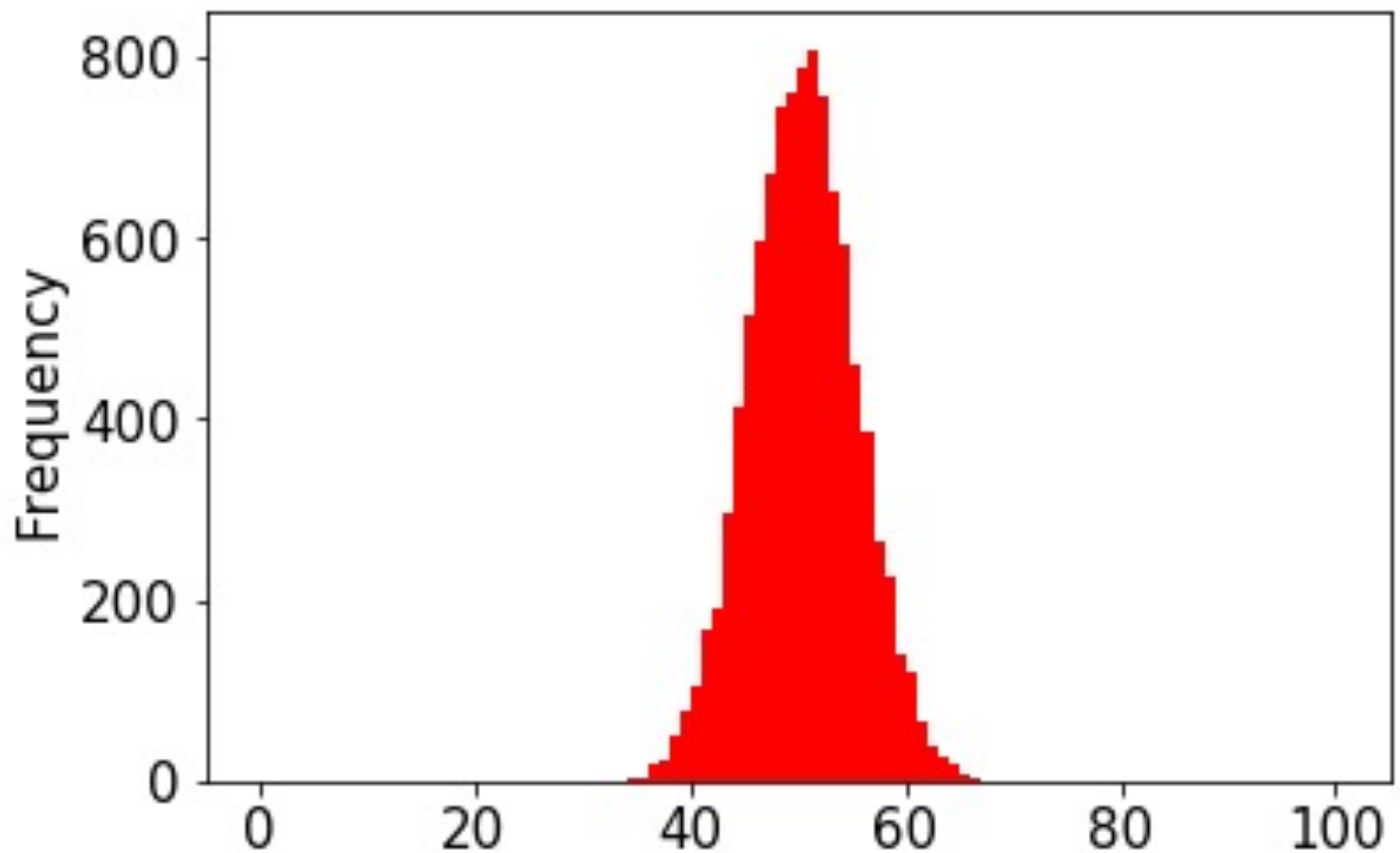
histograms are a good way to view frequencies across numbers

```
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



number of heads (out of 100)

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())
```

Output:

```
-0.18638553993371157
0.02888452916769247
1.247456111372642
-0.5388224399358179
-0.45143322136388525
-1.4001861112018241
0.28119371511868047
0.2608861898556597
-0.19246288728955144
0.297957296171029
```

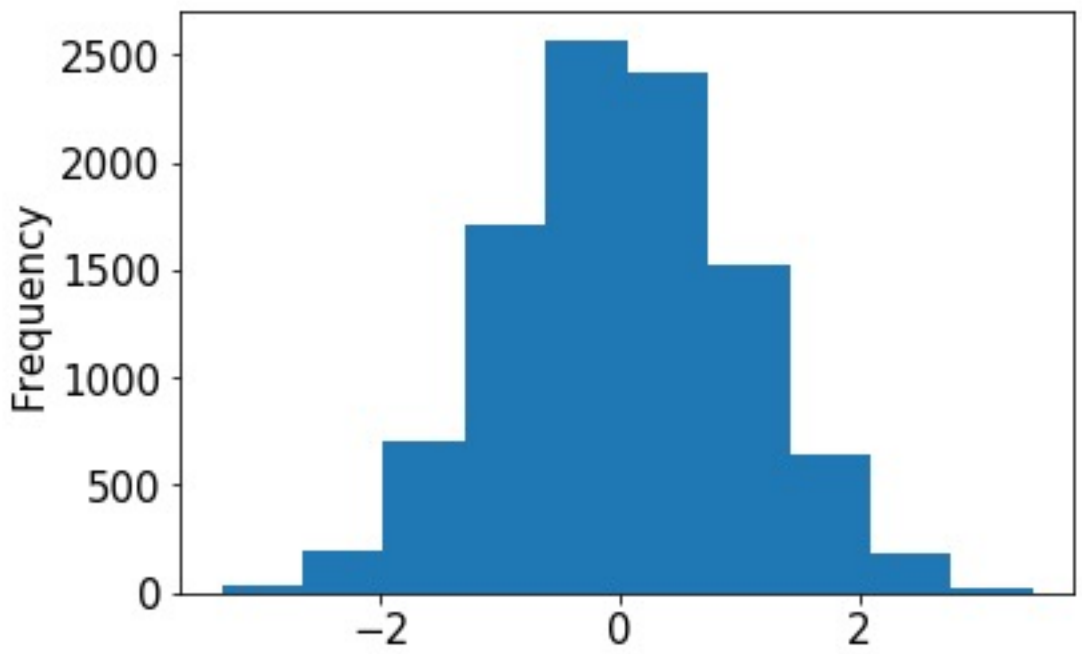
average is 0 (over many calls)

numbers closer to 0 more likely

-x just as likely as x

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)
```