LIST ACCESS, HASHING, SIMULATIONS, & WRAP-UP!

(download slides and .py files to follow along)

6.100L Lecture 26

Ana Bell

TODAY

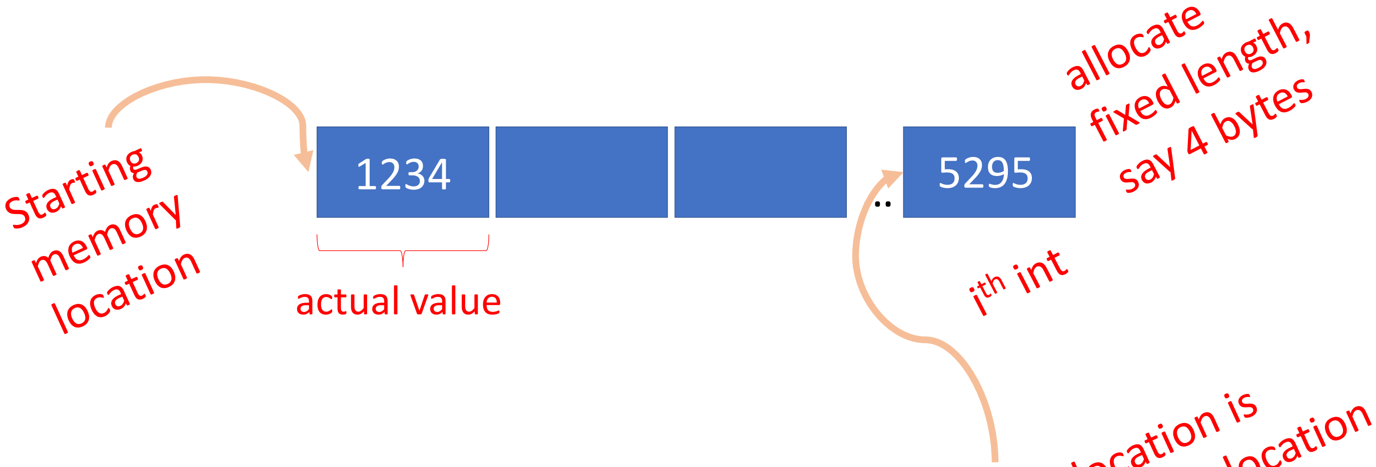
- A bit about lists
- Hashing
- Simulations

LISTS

COMPLEXITY OF SOME PYTHON OPERATIONS

- Lists: `n` is `len(L)`
  - access θ(1)
  - store θ(1)
  - length θ(1)
  - append θ(1)
  - **== θ(n)**
  - **delete θ(n)**
  - **copy θ(n)**
  - **reverse θ(n)**
  - **iteration θ(n)**
  - **`in` list θ(n)**

CONSTANT TIME LIST ACCESS



Starting memory location

actual value

allocate fixed length, say 4 bytes

i^{th} int

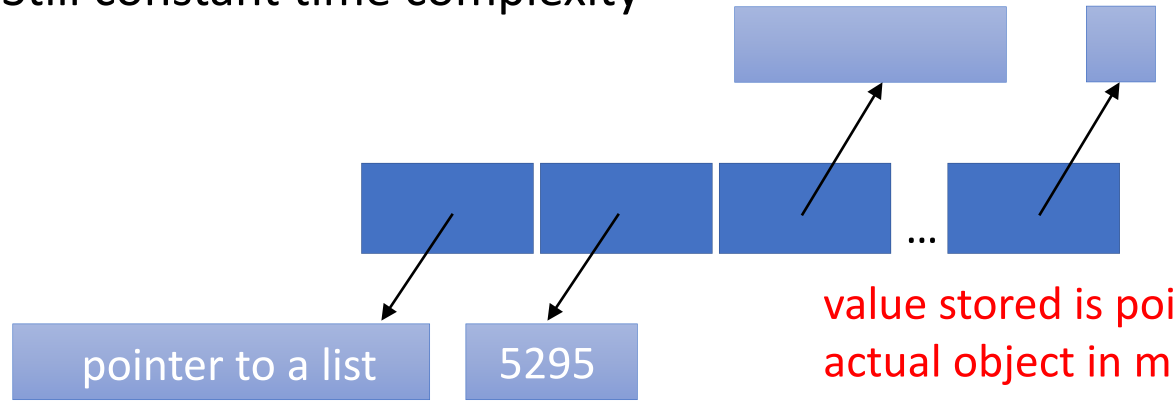
This location is $32*i$ + start location

- If list is all `ints`, list of length L
  - Set aside 4*len(L) bytes
  - Store values directly
  - Consecutive set of memory locations
- List name points to first memory location
- To access i^{th} element
  - **Add 32*i to first location**
  - Access that location in memory
  - **Constant** time complexity

since entries are 4*8=32 bits long

CONSTANT TIME LIST ACCESS

- If list is **heterogeneous**
  - Can’t store values directly (don’t all fit in 32 bits)
  - Use **indirection** to reference other objects
  - **Store pointers** to values (not value itself)
  - Still use consecutive set of memory locations
  - Still set aside 4*len(L) bytes
  - Still add 32*i to first location and +1 to access that location in memory
  - Still constant time complexity



value stored is pointer to actual object in memory

still allocate fixed length units

NAÏVE IMPLEMENTATION OF dict

- Just use a list of pairs: key, value

```
[['Ana', True], ['John', False], ['Eric', False], ['Sam', False]]
```

- What is time **complexity to index** into this naïve dictionary?
  - We don’t know the order of entries
  - Have to do **linear search** to find entry

COMPLEXITY OF SOME PYTHON OPERATIONS

- Lists: `n` is `len(L)`
  - access θ(1)
  - store θ(1)
  - length θ(1)
  - append θ(1)
  - **== θ(n)**
  - **delete θ(n)**
  - **copy θ(n)**
  - **reverse θ(n)**
  - **iteration θ(n)**
  - **in list θ(n)**

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - **length θ(n)**
  - **access θ(n)**
  - **store θ(n)**
  - **delete θ(n)**
  - **iteration θ(n)**
- average case
  - access θ(1)
  - store θ(1)
  - delete θ(1)
  - in θ(1)
  - **iteration θ(n)**

Why?

HASHING

DICTIONARY IMPLEMENTATION

- Uses a **hash table**
- How it does it
  - Convert **key to an integer** – use a **hash function**
  - Use that integer as the index into a list
    - This is constant time
  - Find value associated with key
    - This is constant time
- Dictionary lookup is **constant time complexity**
  - If hash function is fast enough
  - If indexing into list is constant

QUERYING THE HASH FUNCTION

- Just to reveal what’s under the hood, a function `hash()`

```
In [9]: hash(123)
Out[9]: 123

In [10]: hash("6.100L is awesome")
Out[10]: 8708784260240907980

In [11]: hash((1,2,3))
Out[11]: 529344067295497451

In [12]: hash([1,2,3])
Traceback (most recent call last):

  File "<ipython-input-12-35e31e935e9e>",
line 1, in <module>
    hash([1,2,3])

TypeError: unhashable type: 'list'
```

May vary because Python adds randomness to thwart attacks

Why do this? Because hashing is also used to encrypt data for safe storage and retrieval.

HASH TABLE

- **How big** should a hash table be?
- To avoid many keys hashing to the same value, have each key hash to a separate value
- If hashing strings:
  - Represent **each character** with binary code
  - **Concatenate bits together**, and **convert to an integer**

NAMES TO INDICES

- E.g., `'Ana Bell'`

  `= 01000001 01101110 01100001 00100000 01000010 01100101 01101100 01101100`

  `= 4,714,812,651,084,278,892`
- **Advantage**: unique names mapped to **unique indices**
- **Disadvantage**: VERY **space inefficient**
- Consider a table containing MIT's ~4,000 undergraduates
  - Assume longest name is 20 characters
  - Each character 8 bits, so 160 bits per name
  - How many entries will table have?

2^{160}

1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976

A BETTER IDEA: ALLOW COLLISIONS

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a hash table with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

Ana	C

5 + 18 + 9 + 3 = 35
35%16 = 3

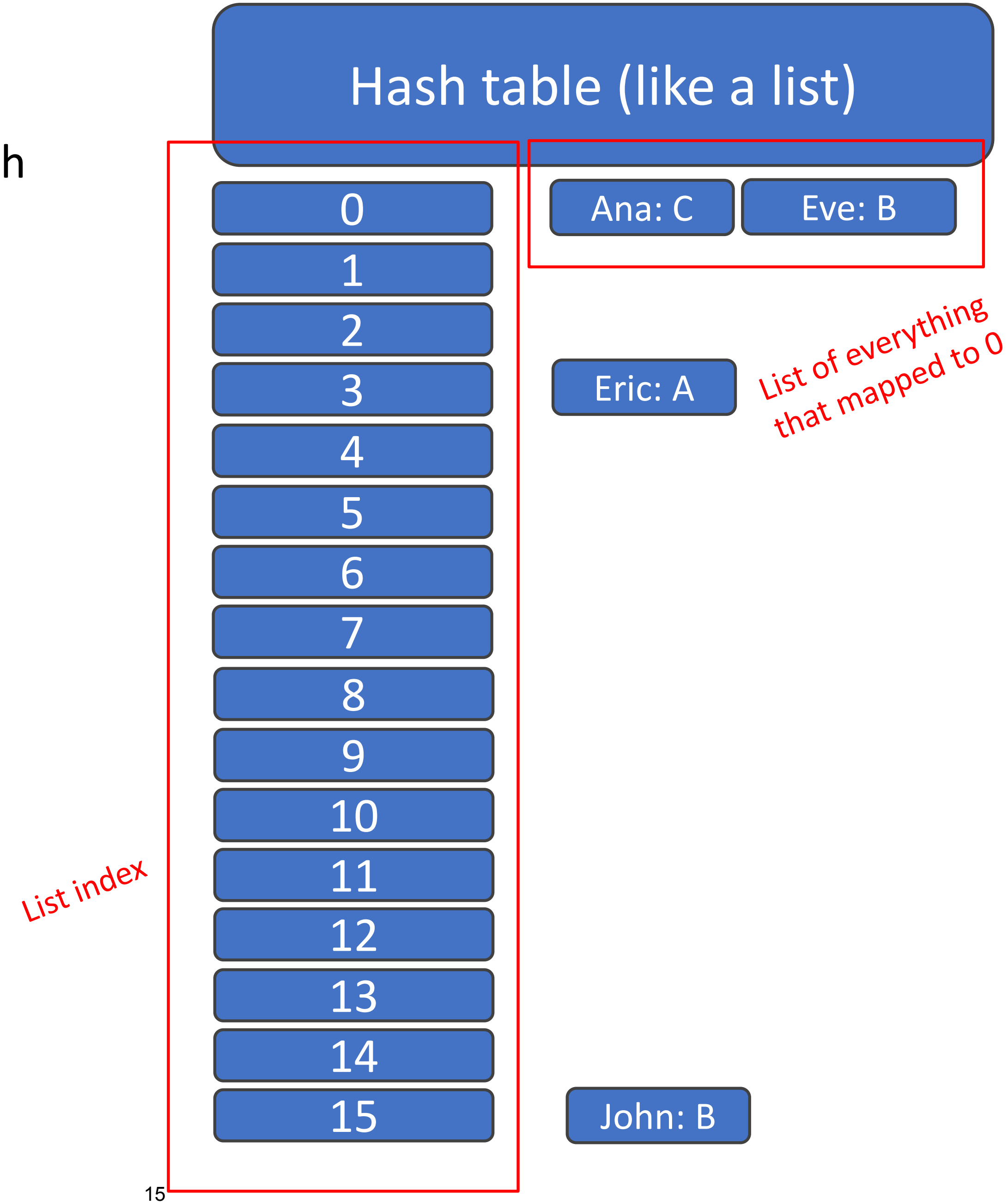
Eric	A

10 + 15 + 8 + 14 = 47
47%16 = 15

John	B

5 + 22 + 5 = 32
32%16 = 0

Eve	B



List of everything that mapped to 0

List index

PROPERTIES OF A GOOD HASH FUNCTION

- Maps domain of interest to integers between 0 and size of hash table
- The hash value is **fully determined by value** being hashed (nothing random)
- The hash function uses the entire input to be hashed
  - Fewer collisions
- **Distribution of values is uniform**, i.e., equally likely to land on any entry in hash table
- Side Reminder: keys in a dictionary must be **hashable**
  - aka immutable
  - They always hash to the same value
  - What happens if they are not hashable?

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

Ana | C

5 + 18 + 9 + 3 = 35
35%16 = 3

Eric | A

10 + 15 + 8 + 14 = 47
47%16 = 15

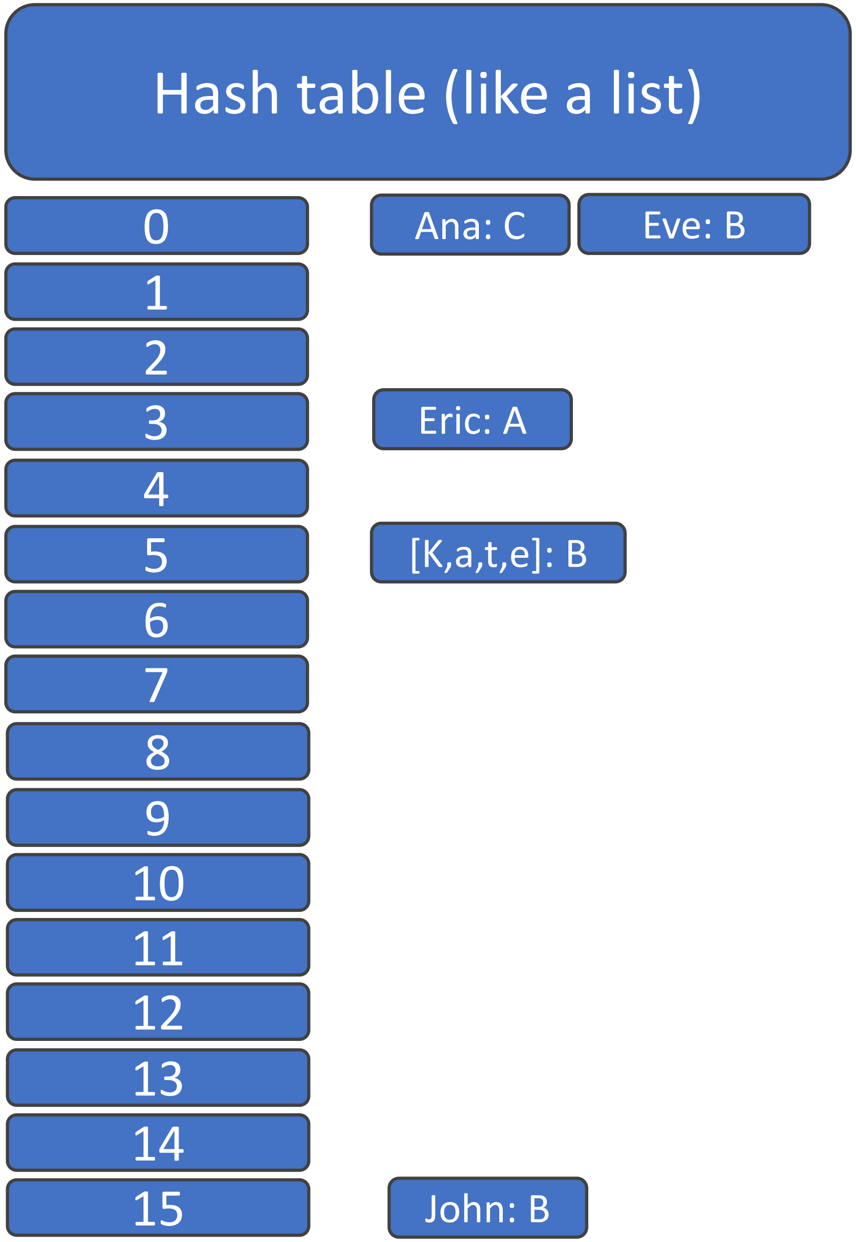
John | B

5 + 22 + 5 = 32
32%16 = 0

Eve | B

11 + 1 + 20 + 5 = 37
37%16 = 5

[K, a, t, e] | B



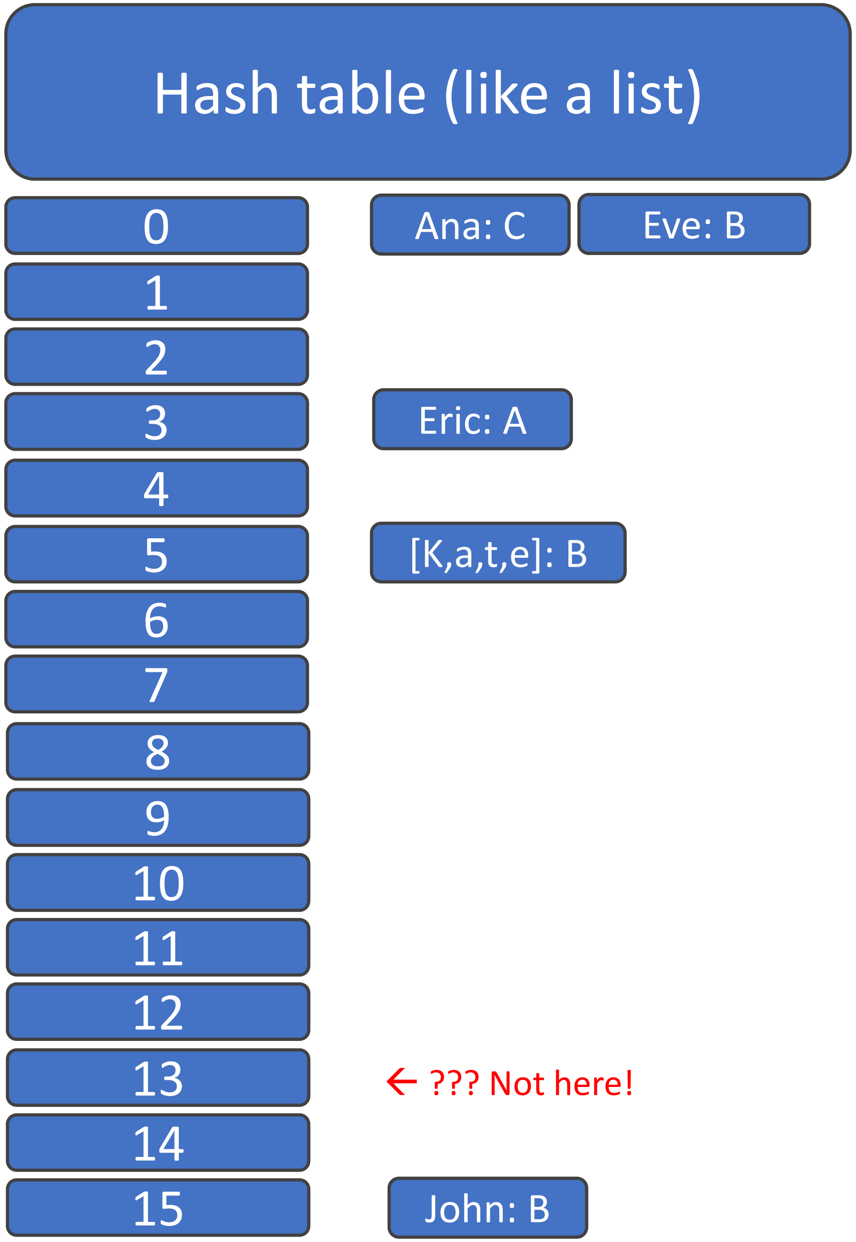
Hash table (like a list)

0 Ana: C Eve: B
1
2
3 Eric: A
4
5 [K,a,t,e]: B
6
7
8
9
10
11
12
13
14
15 John: B

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

Kate changes her name to Cate. Same person, different name. Look up her grade?



Hash table (like a list)

0 Ana: C Eve: B

1

2

3 Eric: A

4

5 [K,a,t,e]: B

6

7

8

9

10

11

12

13 ← ??? Not here!

14

15 John: B

3 + 1 + 20 + 5 = 29

29%16 = 13

[C, a, t, e]

COMPLEXITY OF SOME PYTHON OPERATIONS

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - **length θ(n)**
  - **access θ(n)**
  - **store θ(n)**
  - **delete θ(n)**
  - **iteration θ(n)**
- average case
  - access θ(1)
  - store θ(1)
  - delete θ(1)
  - in θ(1)
  - **iteration θ(n)**

If all keys hash to the same index

Hash table is large relative to number of keys

Hash function good enough

SIMULATIONS

TOPIC USEFUL FOR MANY DOMAINS

- **Computationally** describe the world using **randomness**
- One very important topic relevant to many fields of study
  - Risk modeling and analysis
  - Reduce complex models
- Idea:
  - **Observe an event** and want to calculate something about it
  - Using computation, **design an experiment** of that event
  - **Repeat the experiment** K many times (make a simulation)
  - **Keep track** of the outcome of your event
  - After K repetitions, **report the value of interest**

ROLLING A DICE

- Observe an event and want to calculate something about it
  - Roll a dice, what’s the **prob to get a ::**? How about a .?
- Using computation, design an experiment of that event
  - Make a **list** representing die faces and **randomly choose one**
  - `random.choice(['.',':',':.','::','::.',':::'])`
- Repeat the experiment K many times (simulate it!)
  - Randomly choose a die face from a list **repeatedly**, 10000 times
  - How? Wrap the simulation in a **loop**!

```
for i in range(10000):
 roll=random.choice(['.',':',':.','::','::.',':::'])
```

- Keep track of the outcome of your event
  - **Count** how many times out of 10000 the roll equaled ::
- After K repetitions, report the value of interest
  - **Divide** the count by 10000

THE SIMULATION CODE

```
def prob_dice(side):
    dice = ['.',':',':.','::','::.',':::']
    Nsims = 10000
    count = 0
    for i in range(Nsims):
        roll = random.choice(dice)
        if roll == side:
            count += 1
    print(count/Nsims)
```

Repeat experiment

Choose random dice face

Count successes

```
prob_dice('.')      0.1677
prob_dice('::')     0.1602
```

THAT’S AN EASY SIMULATION

- We can compute the probability of a die roll mathematically
- **Why bother** with the code?
- Because we can answer **variations** of that original question and we can ask **harder** questions!
  - Small tweaks in code
  - Easy to change the code
  - Fast to run

NEW QUESTION
NOT AS EASY MATHEMATICALLY

- Observe an event and want to calculate something about it
  - Roll a dice 7 times, **what's the prob to get a :: at least 3 times out of 7 rolls**?
- Using computation, design an experiment of that event
  - Make a list representing die faces and **randomly choose one 7 times in a row**
  - **Face counter increments** when you choose :: (keep track of this number)
- Repeat the experiment K many times (simulate it!)
  - **Repeat** the prev step 10000 times.
  - How? Wrap the simulation in a **loop**!
- Keep track of the outcome of your event
  - **Count** how many times out of 10000 the :: face counter >= 3
- After K repetitions, report the value of interest
  - **Divide** the outcome count by 10000

EASY TWEAK TO EXISTING CODE

Generalize fcn

```
def prob_dice_atleast(Nrolls, n_at_least):
    dice = ['.',':',':.','::','::.',':::']
    Nsims = 10000
    how_many_matched = []
    for i in range(Nsims):
        matched = 0
        for i in range(Nrolls):
            roll = random.choice(dice)
            if roll == ':::':
                matched += 1
        how_many_matched.append(matched)

    count = 0
    for i in how_many_matched:
        if i >= n_at_least:
            count += 1
    print(count/len(how_many_matched))
```

Roll 7 times and keep track, in a list, how many :: came up

How many times :: came up >=3 times out of 10000

```
prob_dice_atleast(7, 3)
prob_dice_atleast(1, 1)
```

0.0955

0.16

REAL WORLD QUESTION
VERY COMMON EXAMPLE OF HOW USEFUL SIMULATIONS CAN BE

- Water runs through a faucet somewhere between 1 gallons per minute and 3 gallons per minute
- What's the **time it takes to fill a 600 gallon pool**?
    - Intuition?
    - It's not 300 minutes (600/2)
    - It's not 400 minutes (600/1 + 600/3)/2
- In code:
    - Grab a bunch of random values between 1 and 3
    - Simulate the time it takes to fill a 600 gallon pool with each randomly chose value
    - Print the average time it takes to fill the pool over all these randomly chosen values

```
def fill_pool(size):
    flow_rate = []
    fill_time = []
    Npoints = 10000
    for i in range(Npoints):
        r = 1+2*random.random()
        flow_rate.append(r)
        fill_time.append(size/r)

    print('avg flow_rate:', sum(flow_rate)/len(flow_rate))
    print('avg fill_time', sum(fill_time)/len(fill_time))
    plt.figure()
    plt.scatter(range(Npoints),flow_rate,s=1)
    plt.figure()
    plt.scatter(range(Npoints),fill_time,s=1)

fill_pool(600)
```

How many random values to get

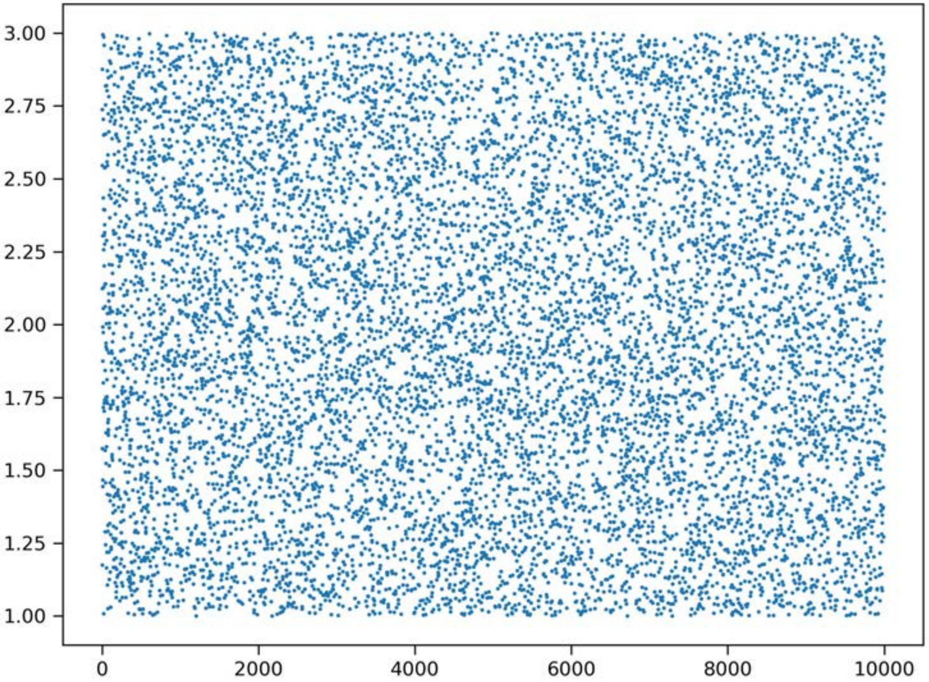
Number between 1 and 3

Use the random value to determine how long it takes to fill up

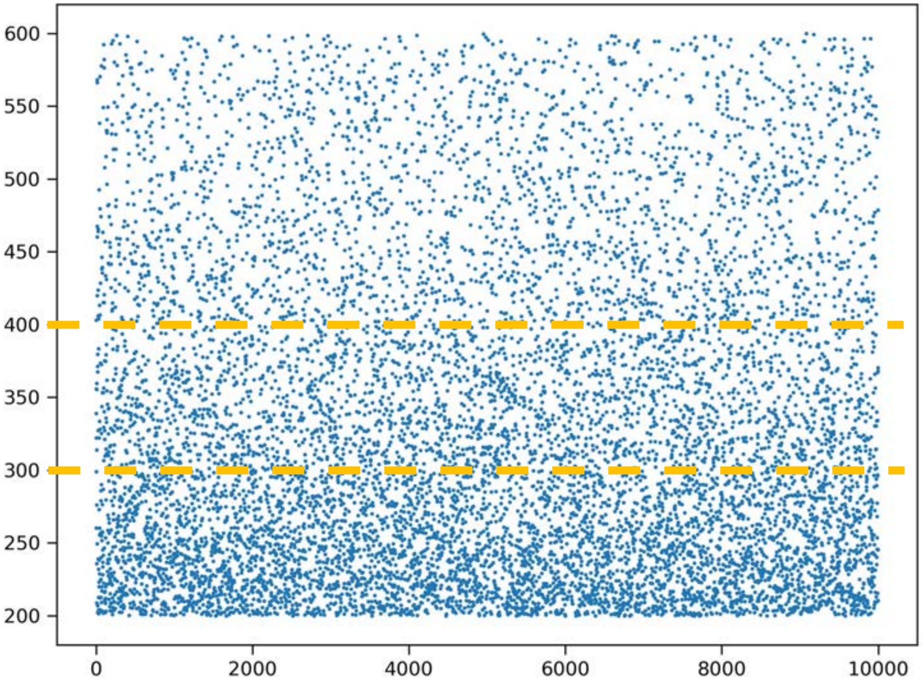
Average over Npoints

PLOTTING RANDOM FILL RATES AND CORRESPONDING TIME IT TAKES TO FILL

Random values for **fill rate**

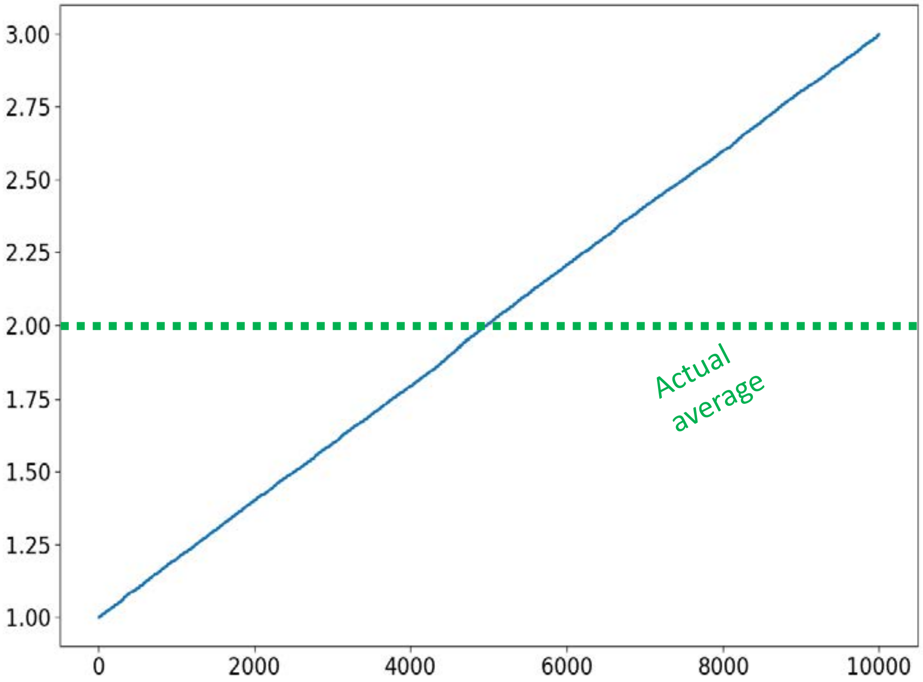


Time to fill using formula pool_size/rate

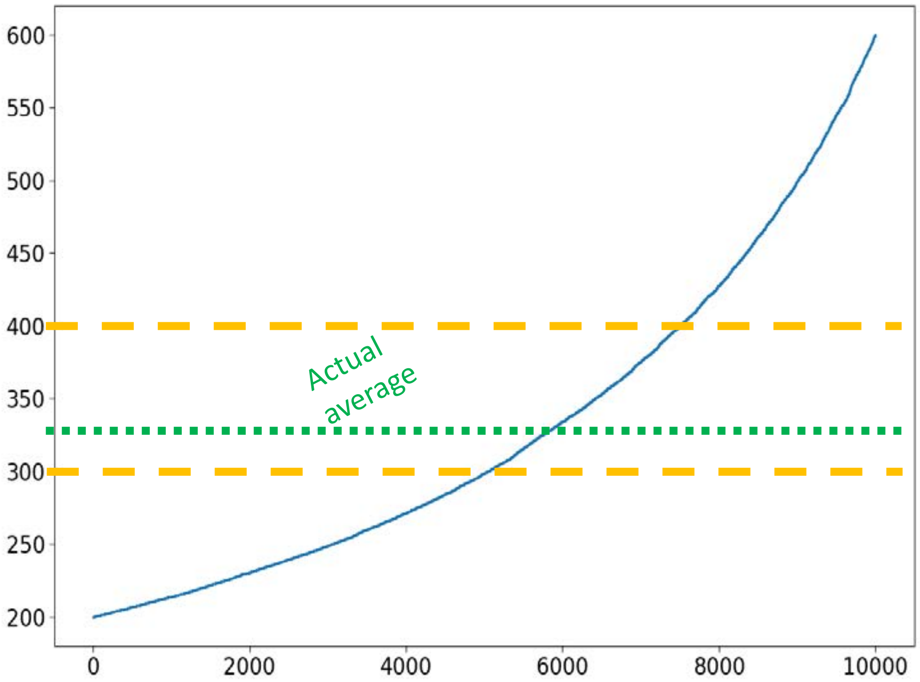


PLOTTING RANDOM FILL RATES AND CORRESPONDING TIME IT TAKES TO FILL

Random values for **fill rate (sorted)**



Time to fill (sorted) using formula pool_size/rate



RESULTS

- avg flow_rate: 1.992586945871106 **approx. 2 gal/min** (avg random values between 1 and 3)
- avg fill_time: 330.6879477596955 **approx. 331 min** (not what we expected!)

- Not 300 and not 400
- There is an **inverse relationship** for fill time vs fill rate
  - Mathematically you'd have to do an **integral**
  - Computationally you just write a **few lines of code**!

WRAP-UP of 6.100L

THANK YOU FOR BEING IN THIS CLASS!

WHAT DID YOU LEARN?

- Python syntax
- Flow of control
  - Loops, branching, exceptions
- Data structures
  - Tuples, lists, dictionaries
- Organization, decomposition, abstraction
  - Functions
  - Classes
- Algorithms
  - Binary/bisection
- Computational complexity
  - Big Theta notation
  - Searching and sorting

YOUR EXPERIENCE

- Were you a “natural”?
- Did you join the class late?
- Did you work hard?

- Look back at the first pset
  it will seem so easy!
- You **learned a LOT** no matter what!

WHAT'S NEXT

- **6.100B** overview of interesting topics in CS and data science (Python)
    - Optimization problems
    - Simulations
    - Experimental data
    - Machine learning

WHAT’S NEXT

- **6.101** fundamentals of programming (Python)
    - Implementing efficient algorithms
    - Debugging

WHAT’S NEXT

- **6.102** software construction (TypeScript)
  - Writing code that is safe from bugs, easy to understand, ready for change

WHAT’S NEXT

- Other classes
  (ML, algorithms, etc.)

IT’S EASY TO FORGET WITHOUT PRACTICE! HAPPY CODING!

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022