Hash Tables

(One of my favorite data structures)

Last Time...

Next Time: Better Solution via Hash-Tables

Hash tables get us a dictionary with..

Set	~O(1)
Insert	~O(1)

Under appropriate conditions

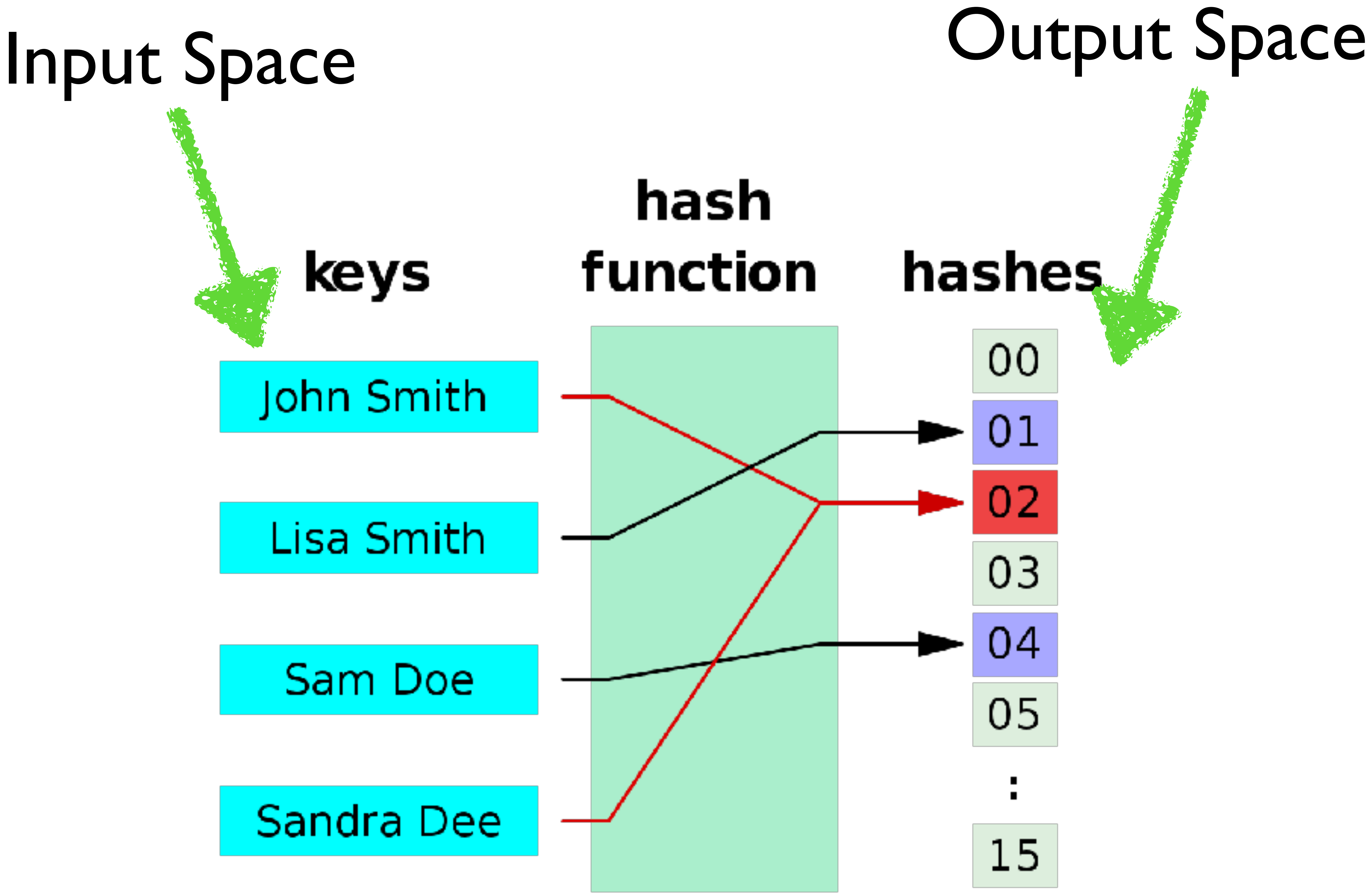
Hash Functions

A **hash function** is a function that takes arbitrarily-length data as input and produces a fixed-length output

You can think of it as "garbling" the data

https://passwordsgenerator.net/sha256-hash-generator/

(There are hundreds of different hash functions, we'll talk about the trade-offs)


Input Space
Output Space
keys
hash
function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

When two **distinct** inputs hash to the same **output**, we call this a **collision**

For example, say your hash function is….

```
f(x) = x % 26
```

- What is the input space?
- What is the output space?
- Find 2 numbers that generate collisions for 13
- Is finding collisions **easy**, or **hard**?

Upshot: This is a crummy hash function

```
f(x) = x % 26
```

Nice properties for hash functions

- Good "dispersal"
  - Things close together hash to things far apart
- Collision-resistant
  - Should be hard to generate a collision
- Non-invertable
  - Should be hard to learn something about input from output

For performance we often just need dispersal, for *security* we often want other two

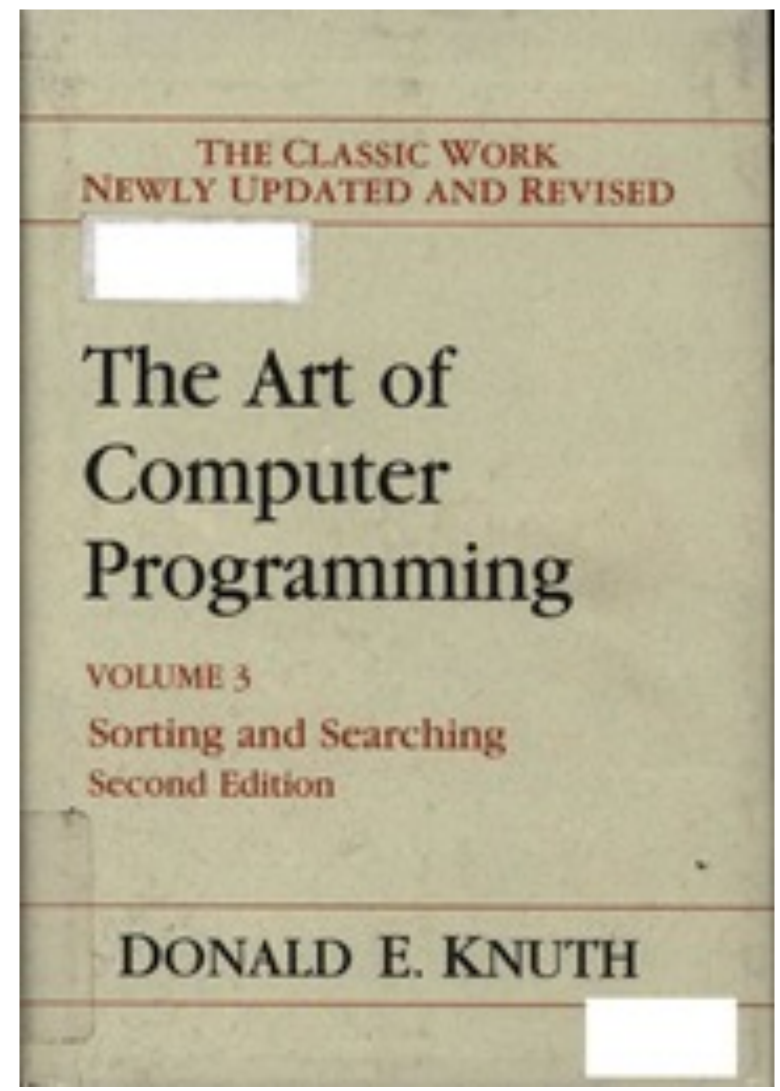
For now, just use Python's built-in
hash()

(If you ever have to do this for real, go read a book)

If you want to hash down to an output space of size n, just do `hash(key) % n`

This is "okay" because builtin hash() has pretty good dispersal properties and modding isn't hurting much

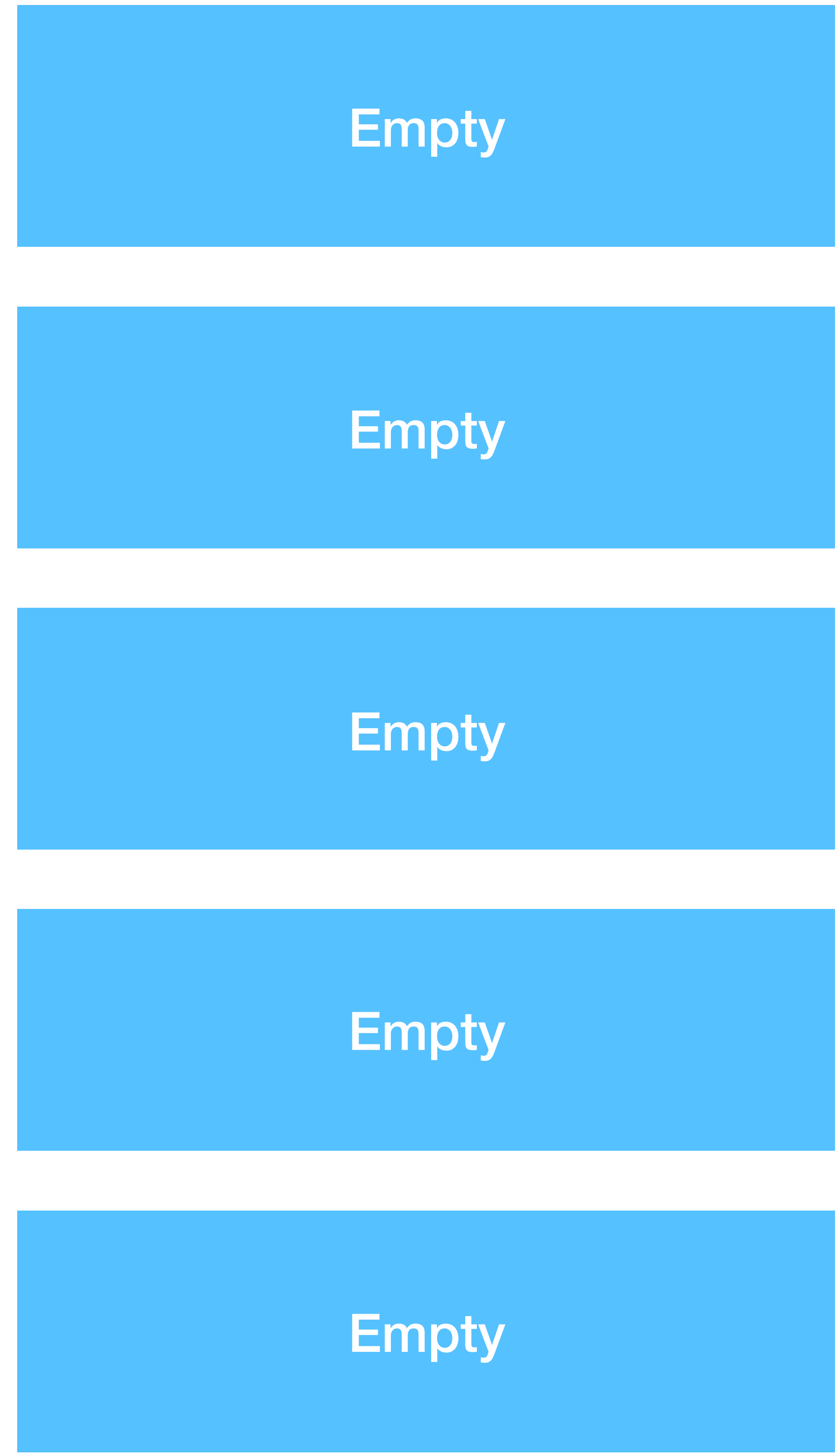
But, again, literally a half of a book about writing good hash functions



The Big Idea

- A hash table is an array of “buckets”
- To store something in table:
  - Hash key, then put value in bucket
- To look up
  - Hash key, go to bucket and find value

An empty hash table is an array of empty buckets



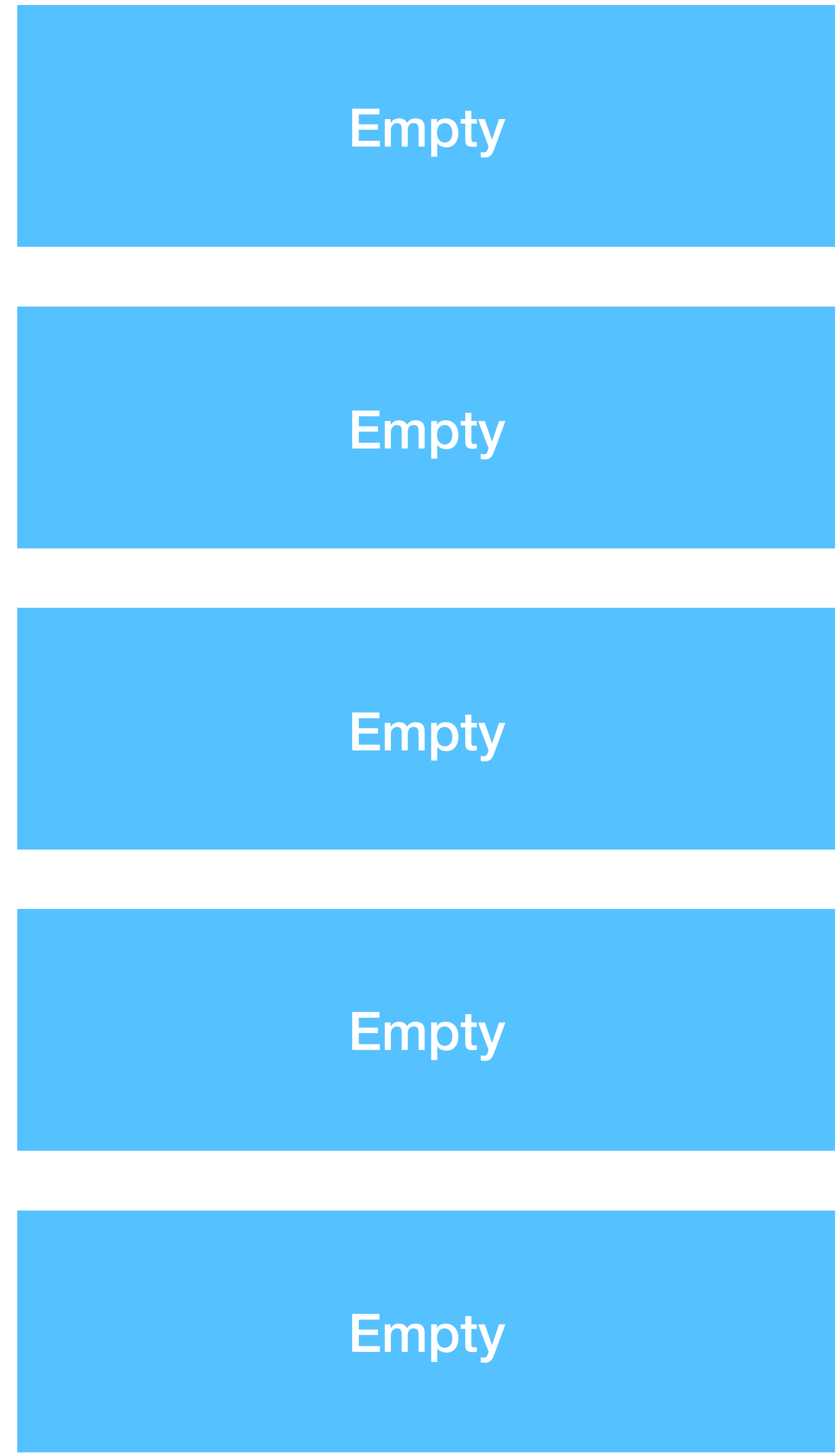
```
class HashTable:
    def __init__(self,numBuckets):
        self.buckets = [None] * numBuckets
        self.numBuckets = numBuckets

    def hash(self,key):
        return hash(key) % self.numBuckets

    def insert(self,key,value):
        ...

    def lookup(self,key):
        ...
```

Let's insert ("Kris", 1990)

Our hash function will be…

```
def myhash(v):
    return hash(v) % 5
```

- Hash key
  - `hash("Kris") % 5 == 0`

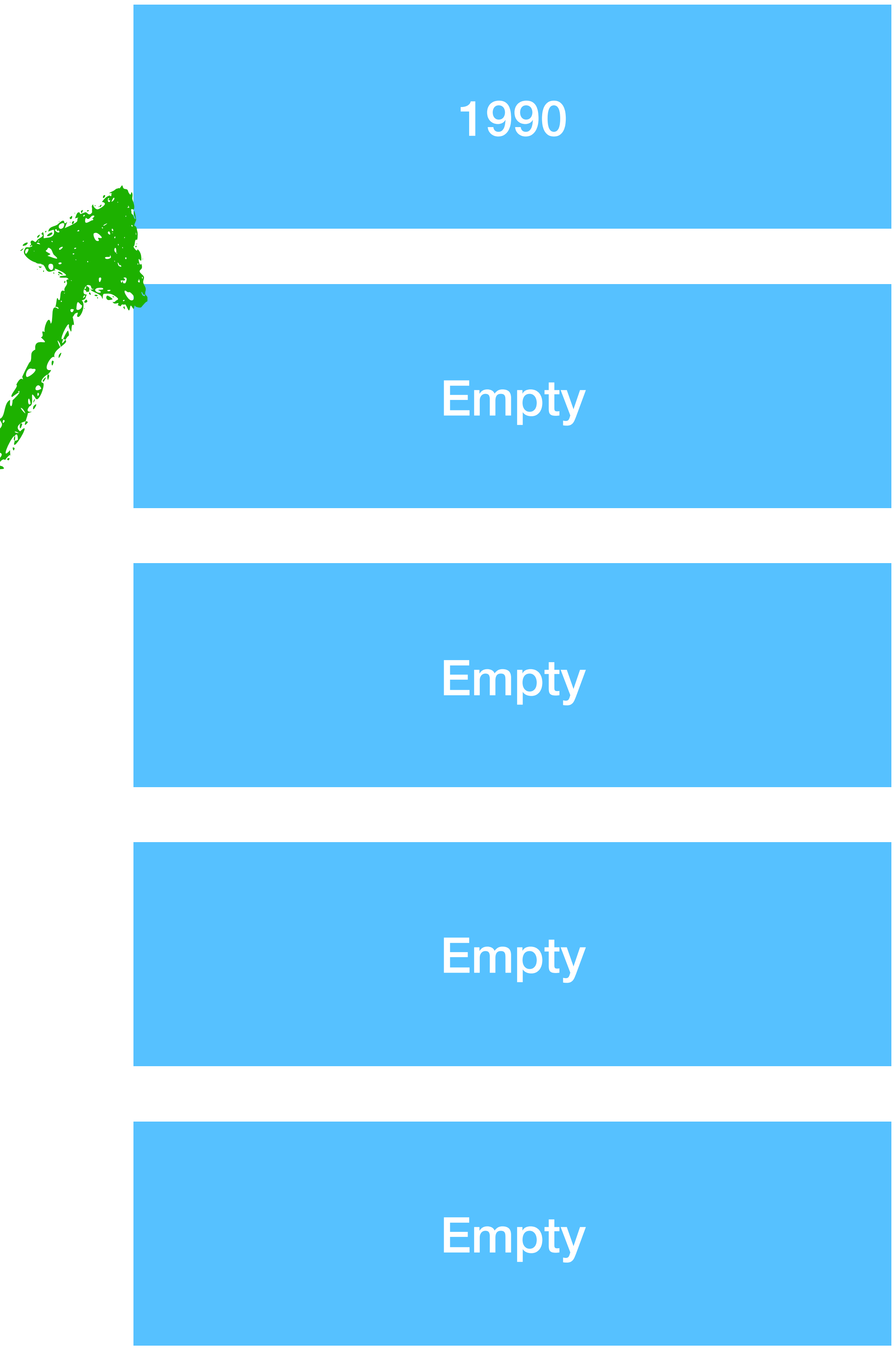


Let's insert ("Kris", 1990)

Our hash function will be…

```
def myhash(v):
    return hash(v) % 5
```

- Hash key
  - `hash("Kris") % 5 == 0`
- Go to 0 and insert 1990



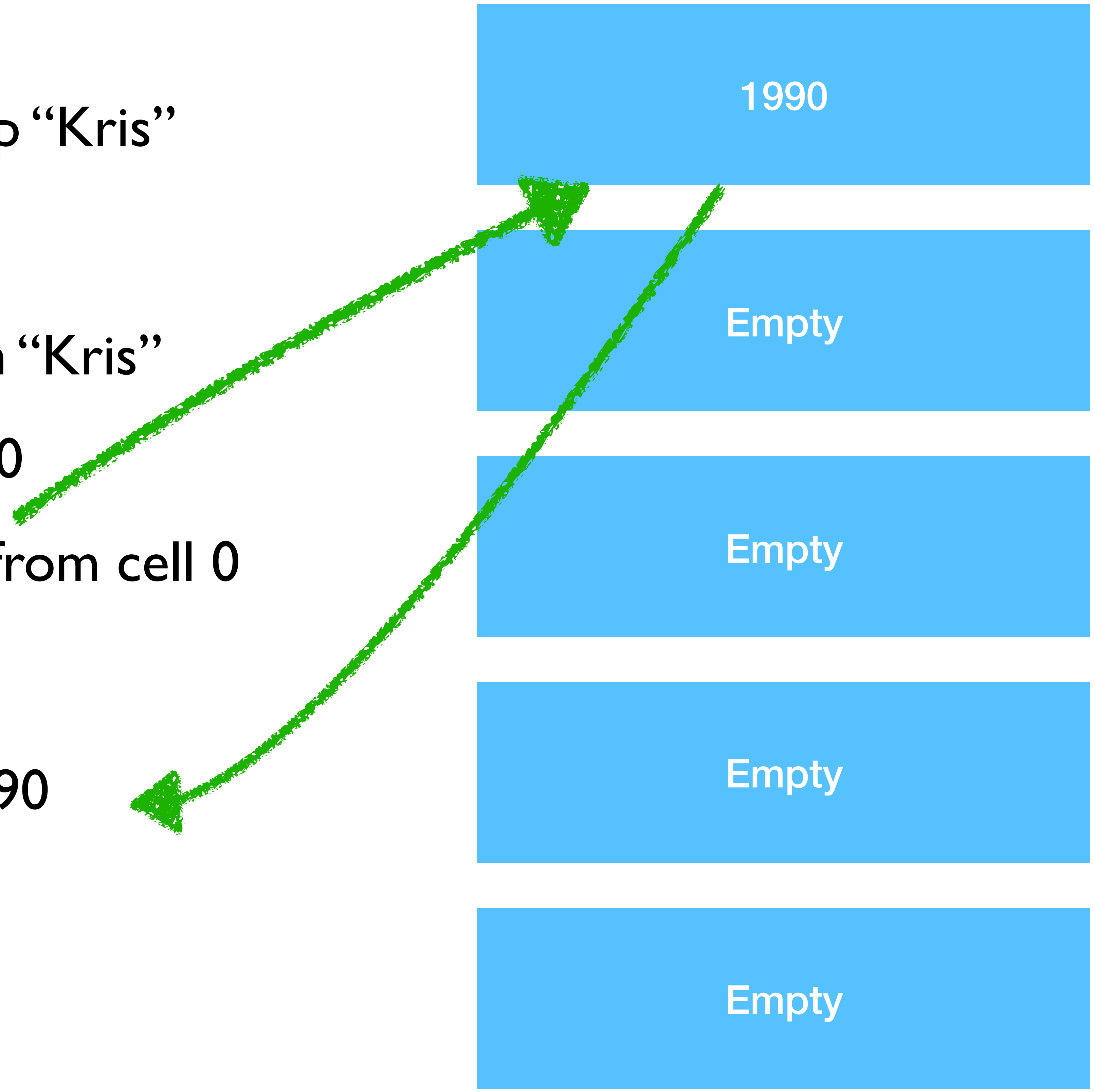
Let’s lookup “Kris”

Again, hash “Kris”

Get 0

Return value from cell 0

Return 1990



Group Challenge

Write `insert` and `lookup`

Then work this example (inserting ("Kris", 1990))

The Problem

This hash table doesn't handle collisions

Challenge

Brainstorm in groups: what can add to work past this problem?

Main Trick

- Back hash-table buckets by association lists
- Works like a hash table until you get to collisions, then works like association list

Group Challenge

Rewrite `insert` and `lookup`

Using association list

(OK to just use regular Python list for now)

Question

Under what circumstance would a hash-table degenerate into a linked list?

Choosing a **Good** hash function

Depends on the application. Do you want:

- Performance (hash fn must be fast)
- Security (need a **cryptographic hash**)
- **Often at odds w/ each other**

Security-Relevant Example

Consider a server that stores all customer account balances in a hash table

Hashing occurs by adding all of the characters of their name and modding by table size

Question: How could you attack this?

Believe it or not, this is **quite a common attack** and most languages do **not** provide cryptographically secure hashes by default!

Examples of cryptographic hashes

MD5 (now broken, collisions can be found in seconds)

SHA-1 (the NSA can break this)

SHA-256 (considered secure, but maybe the NSA can break it)