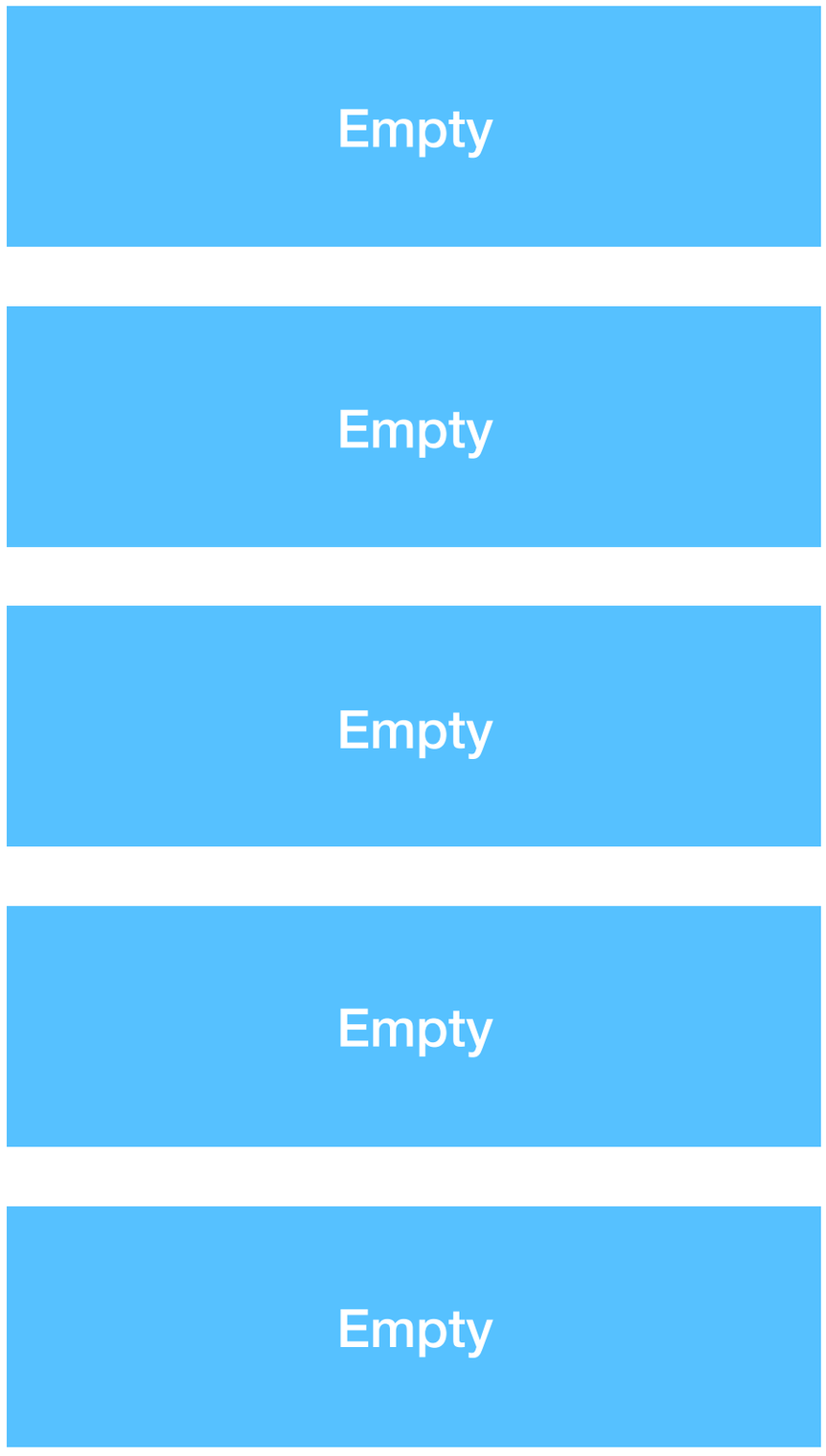
Refreshing last time…

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



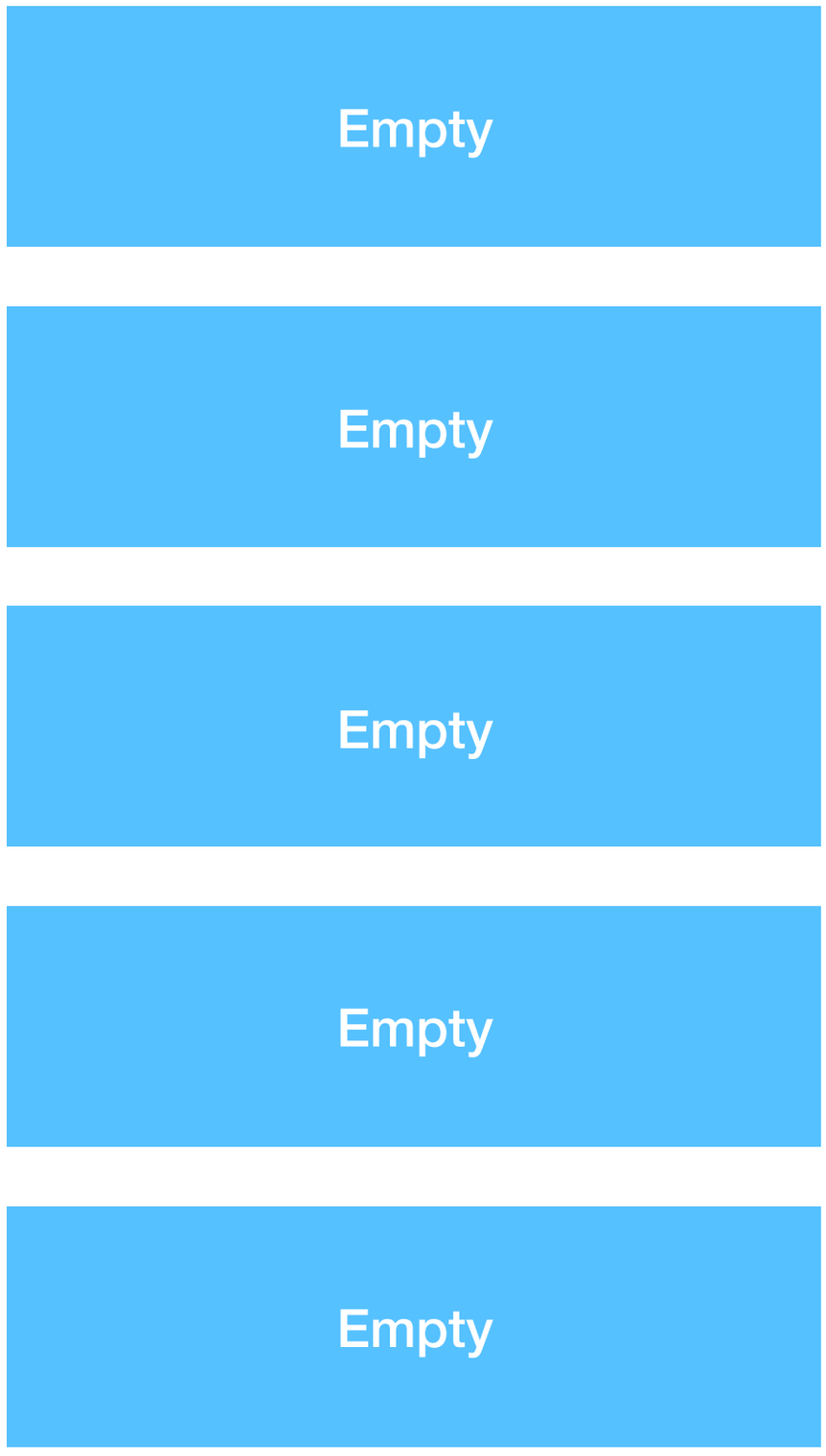
```
class HashTable:
    def __init__(self,numBuckets):
        self.buckets = [None] * numBuckets
        self.numBuckets = numBuckets

    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`

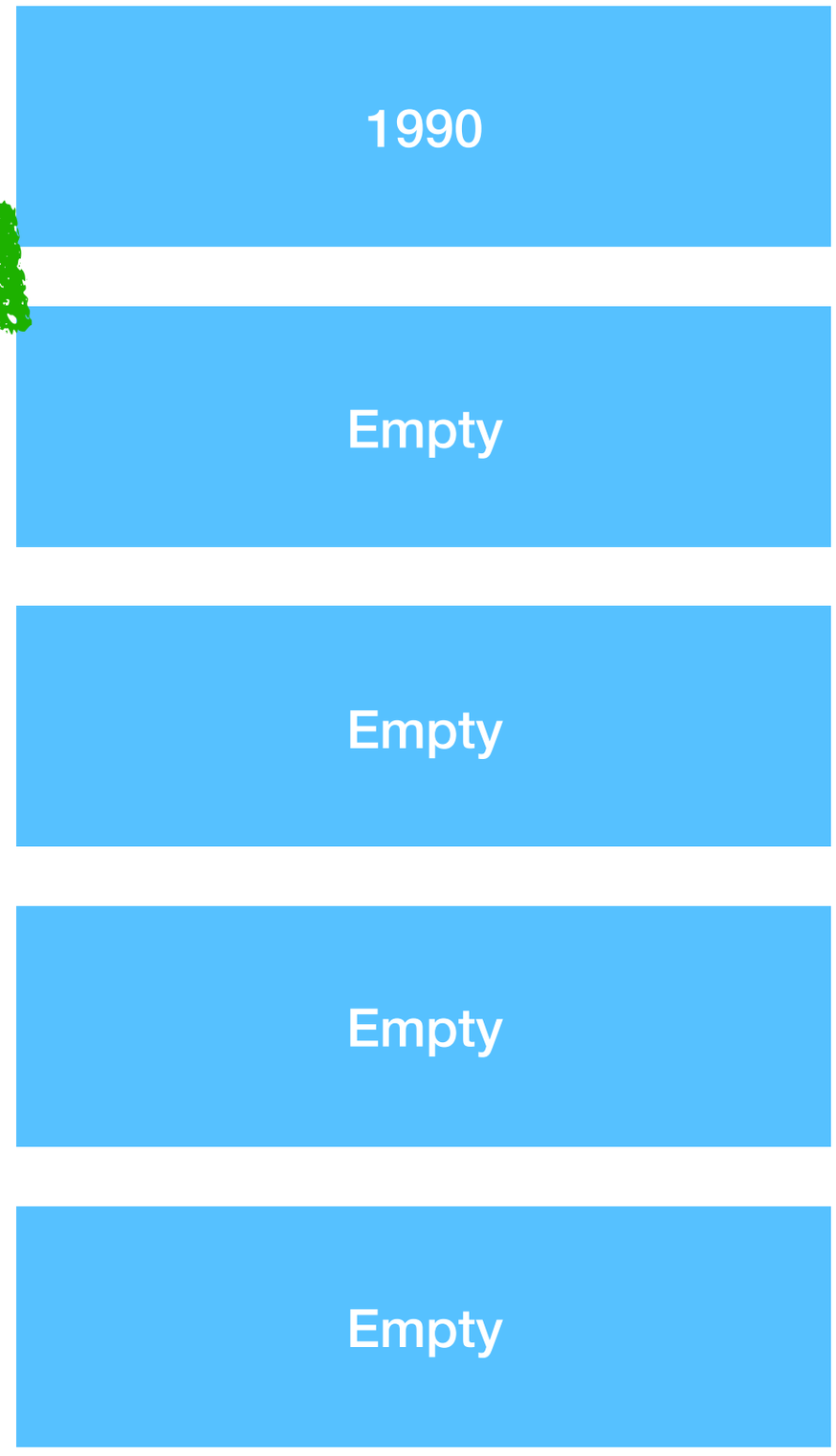


Let's insert ("Kris", 1990)

Our hash function will be…

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

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



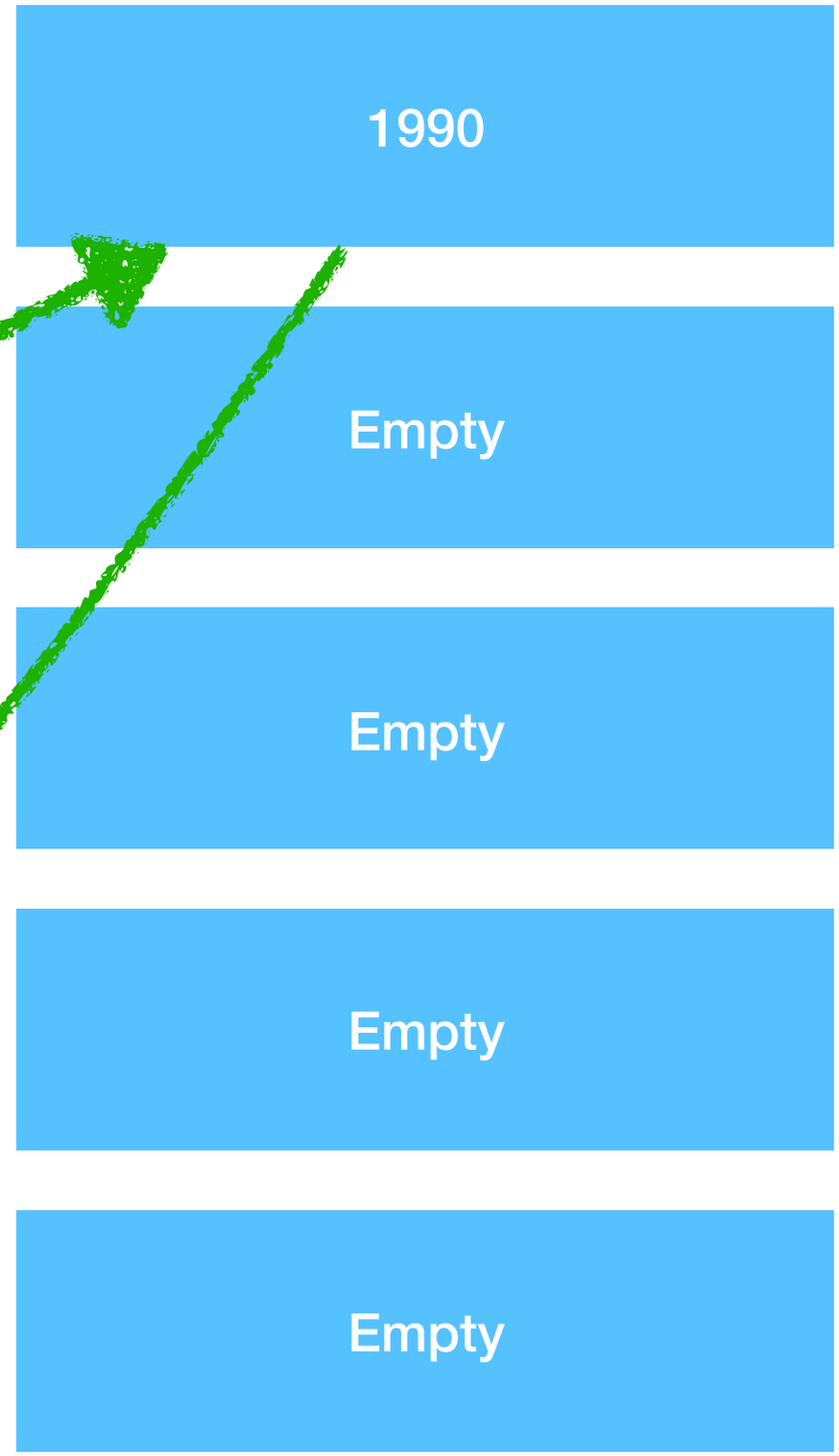
Let's lookup "Kris"

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)

“Shallow” vs. “Deep” Copy

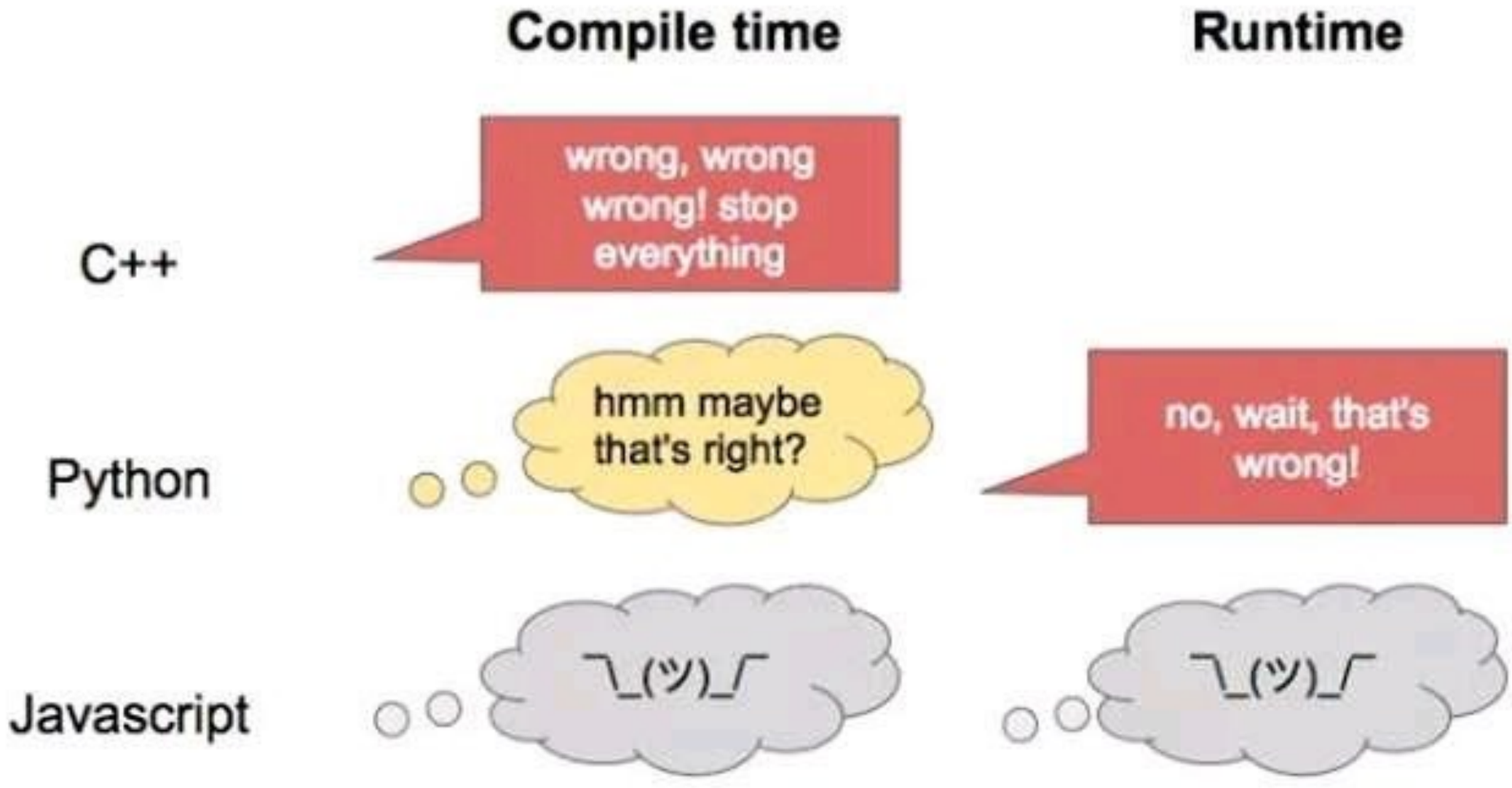
- Some data structures (particularly containers, like lists) store **references** to objects.

- **Upshot**: copies of those data structures will return copies to those objects:

  - E.g., creating a new linked list by allocating new links but reusing the same value maintains reference of value

Subclassing and Polymorphism

When you mistype
`x = obj.fiedl` instead of `x = obj.field`



Polymorphism: the condition of occurring in several different forms

Warmup: Different Tax Rates

- Hypothetical…
  - Clothes are taxed at 18%
  - Food taxed at 4%
    - Health items taxed at 0%
- Task: sum list of items
- Represent as pair:
  - (“clothes”, cost), (“food”, cost), (“health”, cost)

```
itema = ("clothes", 23.50)
itemb = ("food",    14.40)
itemc = ("health",  13.31)

def item_cost(item):
 …
```

```
def sum_items(items):
 x = 0
 for cost in map(item_cost, items):
   x += cost
 return x
```

Challenge: How would you code this up w/ objects..?

```
class Animal:
    def __init__(self,name,species):
        self.name = name
        self.species = species

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def __str__(self):
        return "%s is a %s" % (self.name, self.species)

class Dog(Animal):
    def __init__(self,name):
        self.name = name
        self.species = "canine"
```

```
class Animal:
    def __init__(self,name,species):
        self.name = name
        self.species = species

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def __str__(self):
        return "%s is a %s" % (self.name, self.species)

class Dog(Animal):
    def __init__(self,name):
        self.name = name
        self.species = "canine"
```

Because Dog is a **subclass** of Animal, all of Animal's methods can still be called on a Dog

Draw the fields and (list of) methods
of the following objects…

```
x = Animal("yannis", "giraffe")
y = Dog("ralph")
```

```
class Animal:
    def __init__(self,name,species):
        self.name = name
        self.species = species

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def __str__(self):
        return "%s is a %s" % (self.name, self.species)

class Dog(Animal):
    def __init__(self,name):
        self.name = name

    def getSpecies(self):
        return “canine”
```

- Possible to **override** methods
- Question: if I call getSpecies on an animal object, do I choose Animal or Dog's getSepecies?
- At **runtime** the **most precise** method will be chosen
- Method resolution happens based on the **runtime type**

What does the following print

```
x = Animal("yannis", "giraffe")
y = Dog(“ralph")
print(x.getSpecies())
print(y.getSpecies())
```

What would happen if I used Animal’s getSpecies rather than Dog’s on y?

Method lookup **always** happens based on the **runtime** type of an object

Method chosen will always be **most precise** one for given class

Finding out which method is invoked…

- Say you have a call o.foo(…)
- Step 1: Figure out what the runtime class of o is, say it's O
- Step 2: Go to O's implementation. Look for the method named foo
  - If found, then that's the one that gets called
  - Otherwise, look in O's superclass
  - And so on, until you find one

```
class A:
    def __init__(self):
        self.x = 0
        self.y = 1

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def calculate(self):
        return self.getX() + self.getY()
```

Question: what is A().calculate()?

Tell's Python to run B's **parent's** constructor!

(Since B's parent is A, A.__init__() will be run)

```
class B(A):
    def __init__(self):
        super().__init__()

def getX(self):
    return 2
```

```
class A:
    def __init__(self):
        self.x = 0
        self.y = 1

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def calculate(self):
        return self.getX() + self.getY()
```

```
class B(A):
    def __init__(self):
        super().__init__()

    def getX(self):
        return 2
```

Question: what is B().calculate()?

```
class A:
    def __init__(self):
        self.x = 0
        self.y = 1

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def calculate(self):
        return self.getX() + self.getY()
```

```
class B(A):
    def __init__(self):
        super().__init__()

    def getX(self):
        return 2
```

```
class C(B):
    def __init__(self):
        super().__init__()
        self.y = 3

    def getY(self):
        return 4
```

Question: what is C().calculate()?

Recall: Example from earlier…

```
itema = ("clothes", 23.50)
itemb = ("food",    14.40)
itemc = ("health",  13.31)

def sum_items(items):
 x = 0
 for i in items
   x += item.calculateCost()
```

Key idea: method overloading allows **switching** on object type

```
class A:
  …                class B(A):     class C(A):
  def foo(self): …                   …
  # Case 1
```

```
# Assume o is A or subclass
def foo(o):
  o.foo()
```

Challenge:

Rewrite this code to not use isinstance

Upshot: You should basically ***never*** be using **isinstance**

Almost always indicates **bad style**

Use a polymorphic method instead

(Might need to make new methods)

Challenge: Refactor this code to avoid isinstance

```
class Circle:
    def __init__(self,radius):
        self.radius = radius

class Rectangle:
    def __init__(self,length,width):
        self.length = length
        self.width  = width

def area(o):
    if (isinstance(o,Circle)):
        return (o.radius * o.radius * math.pi)
    elif (isinstance(o,Rectangle)):
        return o.length * o.width
```

Example from HaverQuest