livelessons
video instruction from technology experts

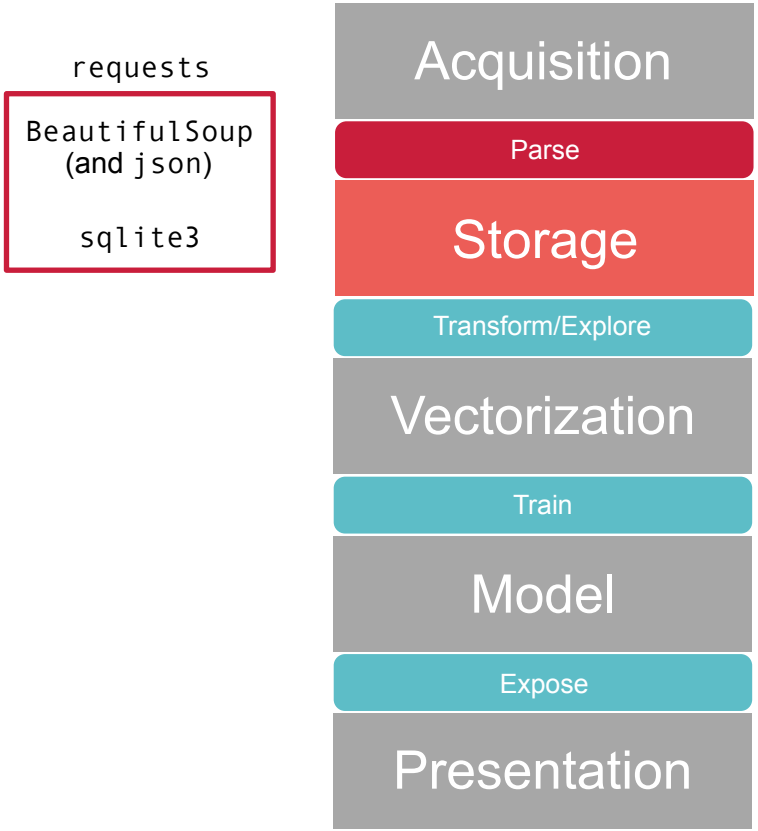
Lesson 4: Adding Structure—Parsing Data and Data Models

For now, we are interested not in listings themselves but the context around the listing

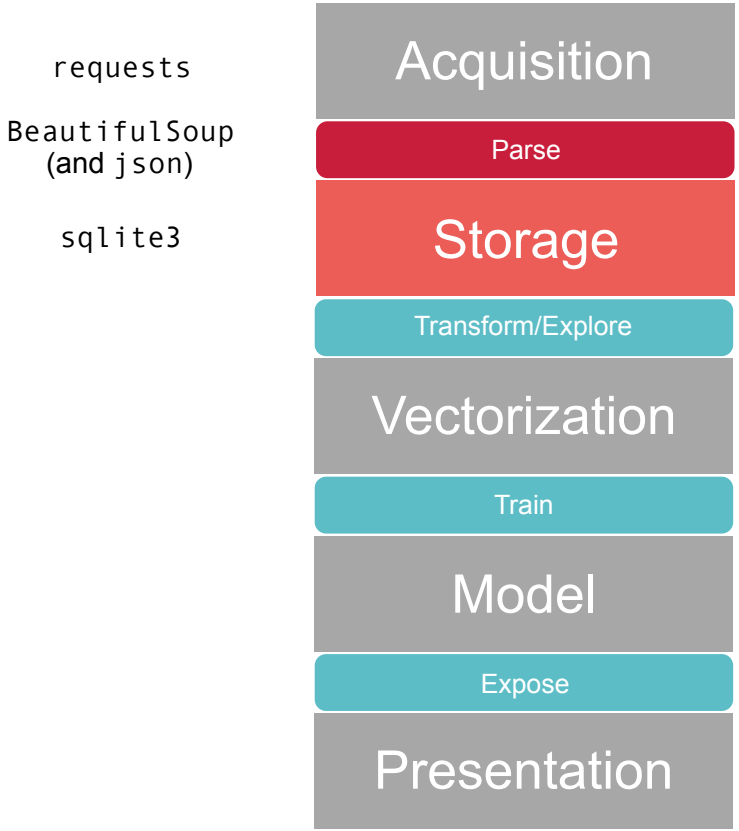
Which neighborhoods are the most explorable?

Which listings have the most lively and active POI around them?

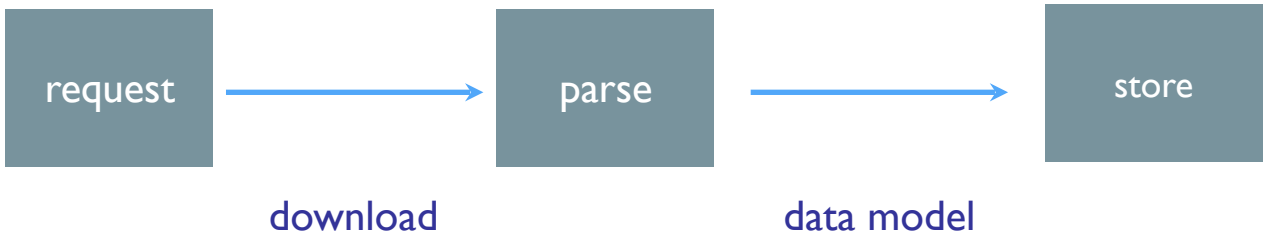
Process + Tools



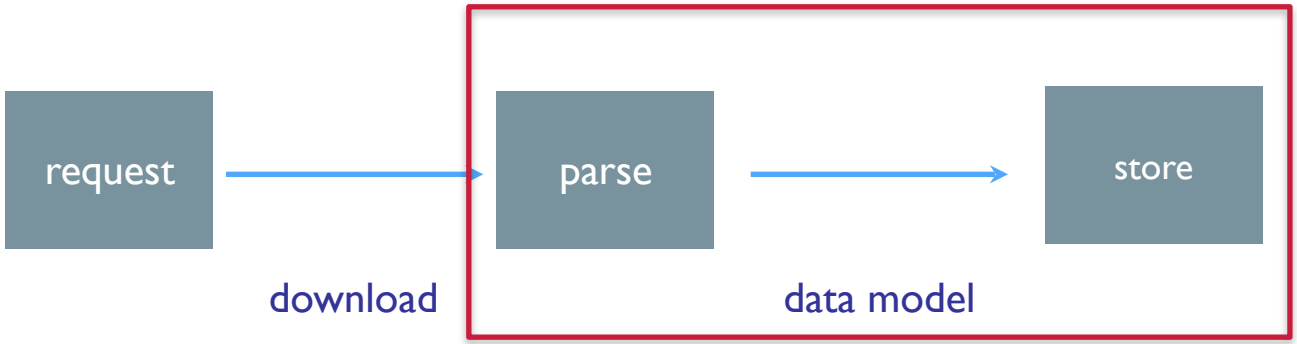
Process + Tools



ETL Pipeline



ETL Pipeline



Overview

Concepts

- Data Models
- Object Oriented Programming
- Schemas and Data Types
- Data Blending and Joining
- Types of Data (Time-series, Geo)
- The Shape of Data (wide vs. long)

Technologies

- Relational Databases (`sqlite3`)
- Object Relational Mappers (`peewee`)

Overview

Ideas

- Data Models
- Object Oriented Programming
- Schemas and Data Types
- Data Blending and Joining
- Types of Data (Time-series, Geo)
- The Shape of Data (wide vs. long)

Implementations

- Relational Databases (`sqlite3`)
- Object Relational Mappers (`peewee`)

ETL Pipeline


```
"venue": {
  "allowMenuUrlEdit": true,
  "categories": [ … ],
  "contact": { … },
  "hasMenu": true,
  "hereNow": { … },
  "hours": { … },
  "id": "4adcd2b6f964a5203d3021e3",
  "location": { … },
  "menu": { … },
  "name": "Grateful Bread Baking Company & Cafe",
  "photos": { … },
  "price": { … },
  "rating": 8.5,
  "ratingColor": "73CF42",
  "ratingSignals": 66,
  "stats": { … },
  "url": "http://gratefulbreadbaking.com",
  "verified": false
}
```

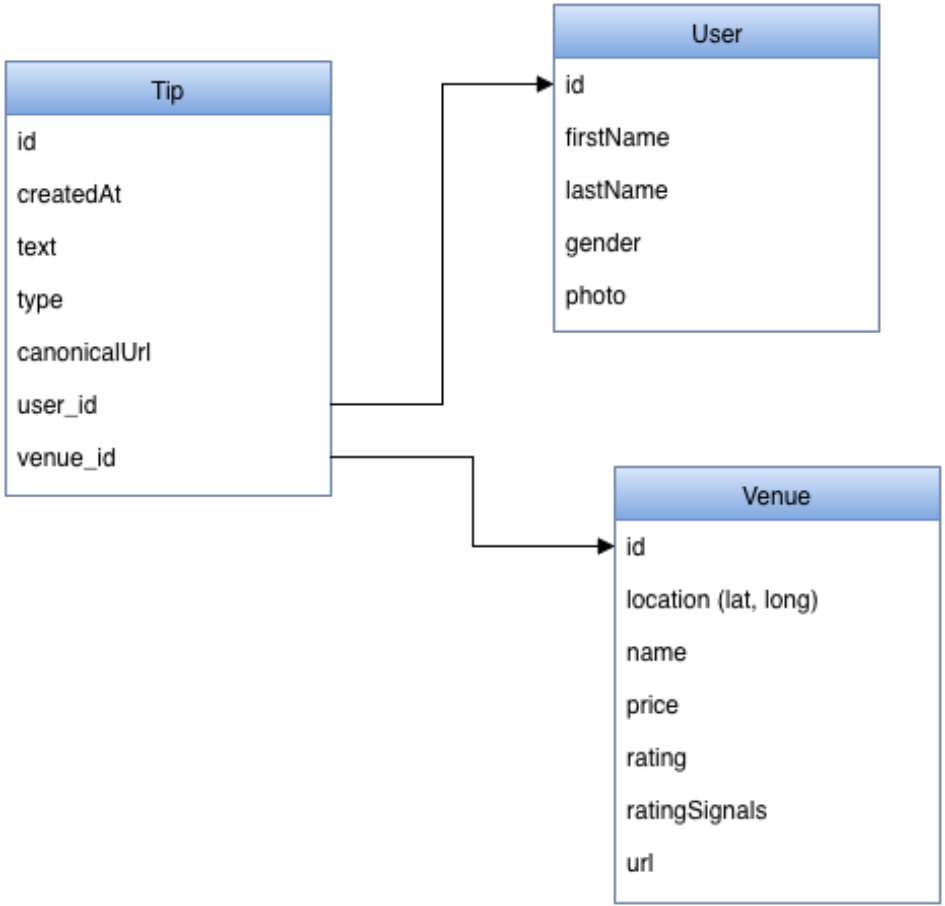

```
"venue": {
  "allowMenuUrlEdit": true,
  "categories": [ … ],
  "contact": { … },
  "hasMenu": true,
  "hereNow": { … },
  "hours": { … },
  "id": "4adcd2b6f964a5203d3021e3",
  "location": { … },
  "menu": { … },
  "name": "Grateful Bread Baking Company & Cafe",
  "photos": { … },
  "price": { … },
  "rating": 8.5,
  "ratingColor": "73CF42",
  "ratingSignals": 66,
  "stats": { … },
  "url": "http://gratefulbreadbaking.com",
  "verified": false
}
```

```
"tips": [
  {
    "canonicalUrl": "https://foursquare.com/item/
4d4d921a9ee1a35d764827df",
    "createdAt": 1296929306,
    "id": "4d4d921a9ee1a35d764827df",
    "likes": {},
    "logView": true,
    "text": "Better food, better coffee and better
staff. Why go to Starbucks down the street and waste
your money?",
    "todo": {},
    "type": "user",
    "user": {
      "firstName": "Marcus",
      "gender": "male",
      "id": "2014328",
      "lastName": "Gorman",
      "photo": {}
    }
  }
]
```

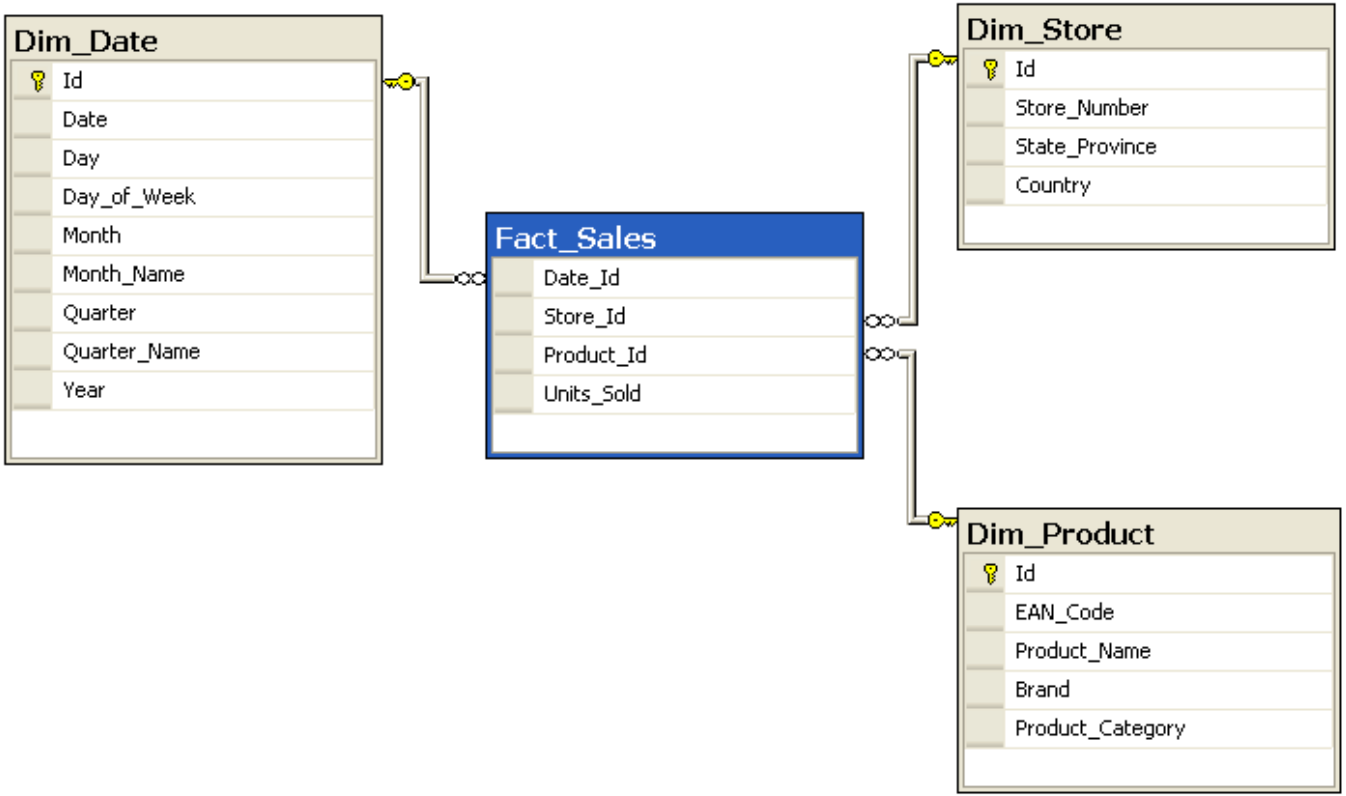

```
  "tips": [
    {
      "canonicalUrl": "https://foursquare.com/item/
4d4d921a9ee1a35d764827df",
      "createdAt": 1296929306,
      "id": "4d4d921a9ee1a35d764827df",
      "likes": {},
      "logView": true,
      "text": "Better food, better coffee and better
staff. Why go to Starbucks down the street and waste
your money?",
      "todo": {},
      "type": "user",
      "user": {
        "firstName": "Marcus",
        "gender": "male",
        "id": "2014328",
        "lastName": "Gorman",
        "photo": {}
      }
    }
  ]
```

Thinking in Tables

Schema



Star Schema



source: https://en.wikipedia.org/wiki/Star_schema

JSON -> Data Tables

Functional Parsing

```
def parse_venue(data):
    return {
        'id': data['id'],
        'name': data['name'],
        'price': data['price']['tier'],
        'rating': data['rating'],
        'ratingSignals': data['ratingSignals'],
        'url': data['url'],
        'latitude' : data['location']['lat'],
        'longitude': data['location']['lng']
    }
```

Functional Parsing

```
def parse_user(data):
    return {
        'id': data['id'],
        'firstName': data['firstName'],
        'lastName': data['lastName'],
        'gender': data['gender'],
        'photo': data['photo']['prefix'] + 'original' + data['photo']['suffix']
    }
```

Functional Parsing

```
def parse_tip(data, whitelist=['id', 'createdAt', 'text', 'type', 'canonicalUrl']):
    user = parse_user(data['user'])
    tip = { key: data[key] for key in whitelist }

    return (tip, user)
```

Functional Parsing

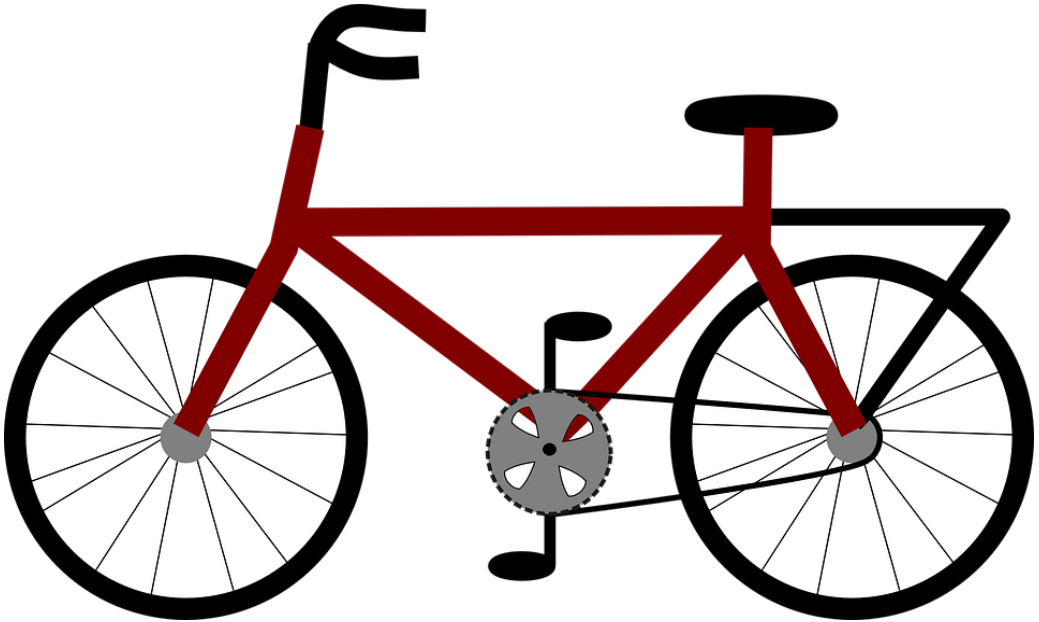
```
def parse_response(data):
    tips = []
    users = {}

    venue = parse_venue(data['venue'])

    for tip in data['tips']:
        tip, user = parse_tip(tip)
        tip['user_id'] = user['id']
        tip['venue_id'] = venue['id']
        tips.append(tip)

        if user['id'] not in users:
            users[user['id']] = user

    return (venue, list(users.values()), tips)
```

Object Oriented Programming

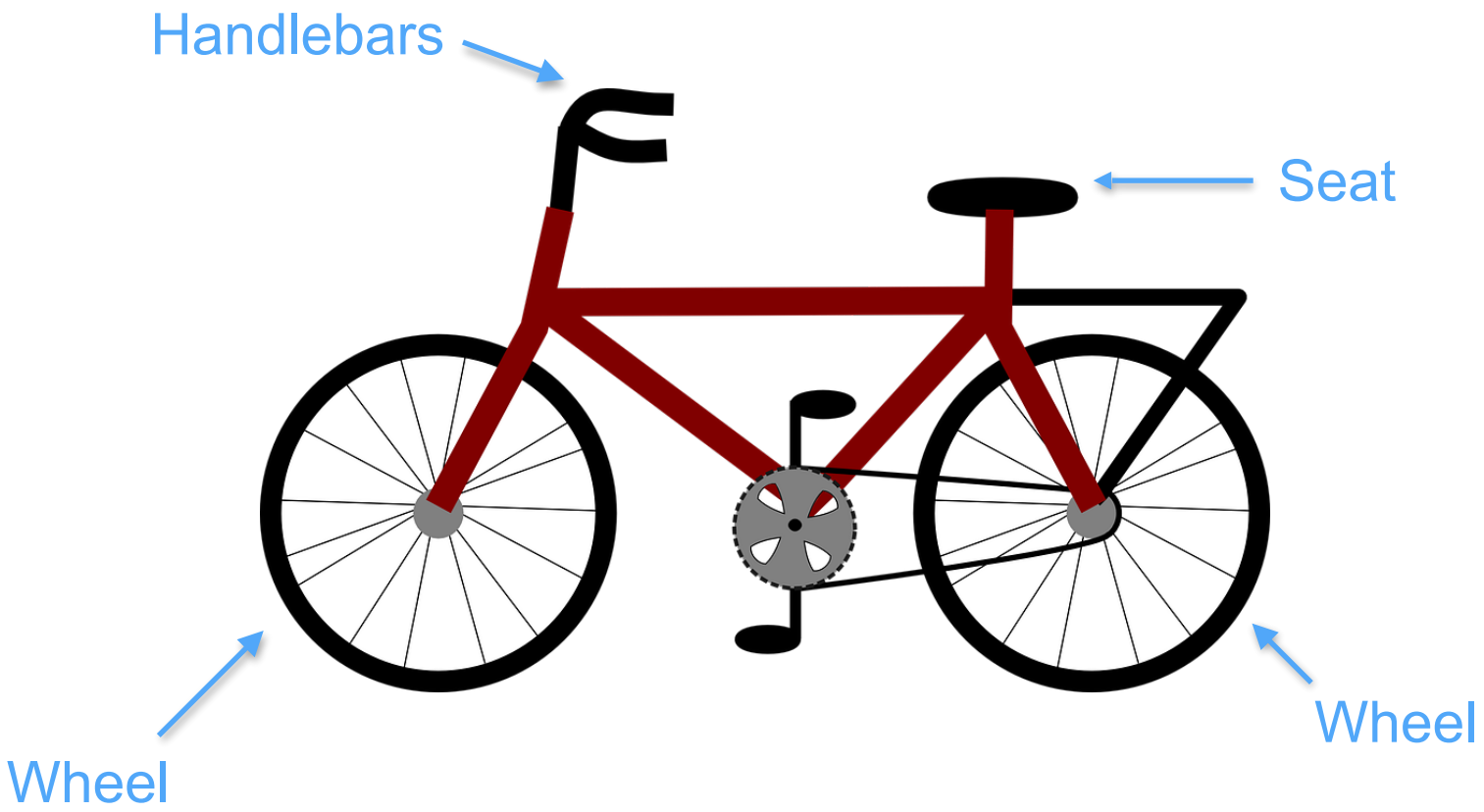
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

- Wikipedia

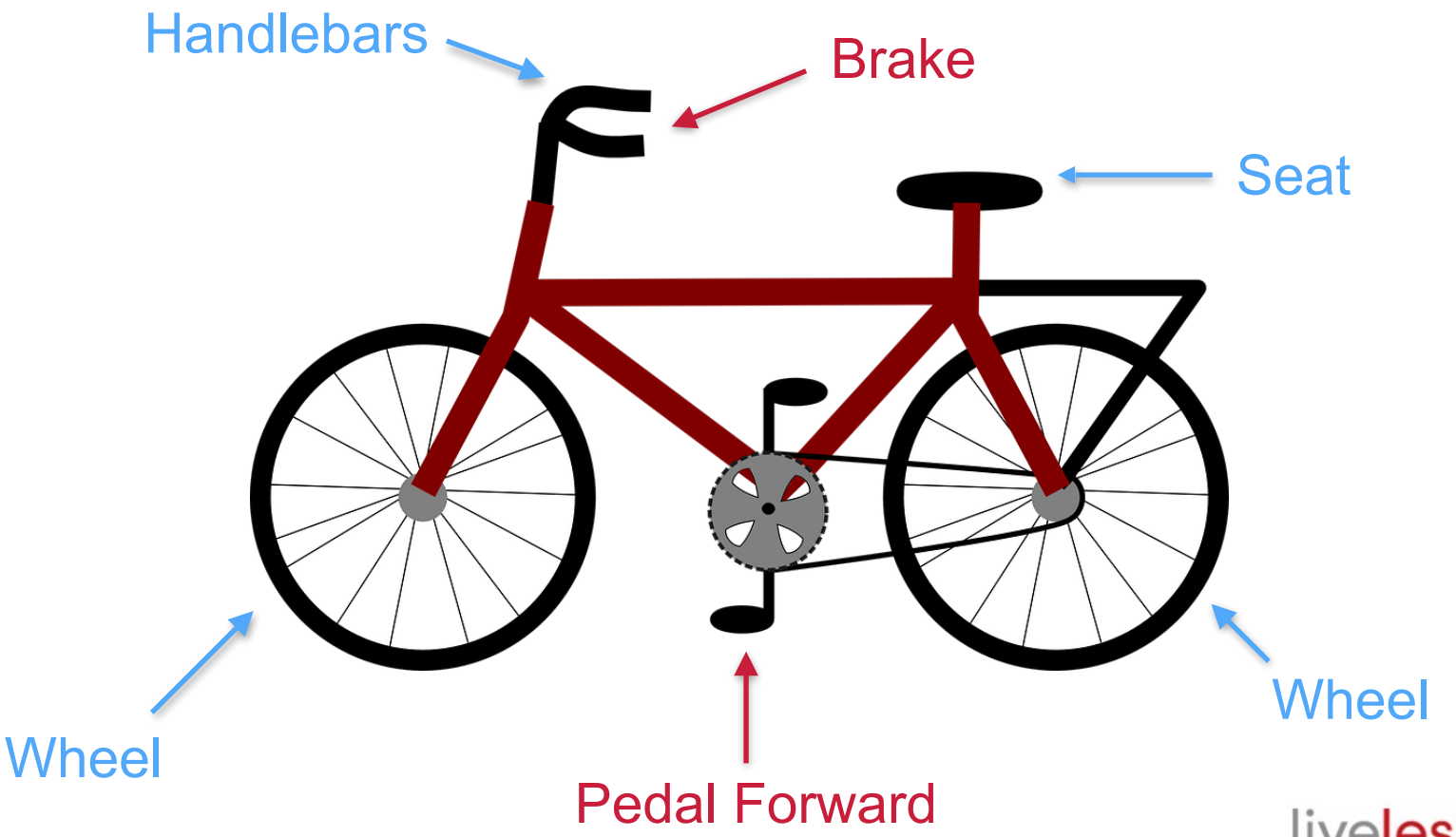
An Object Oriented Bicycle



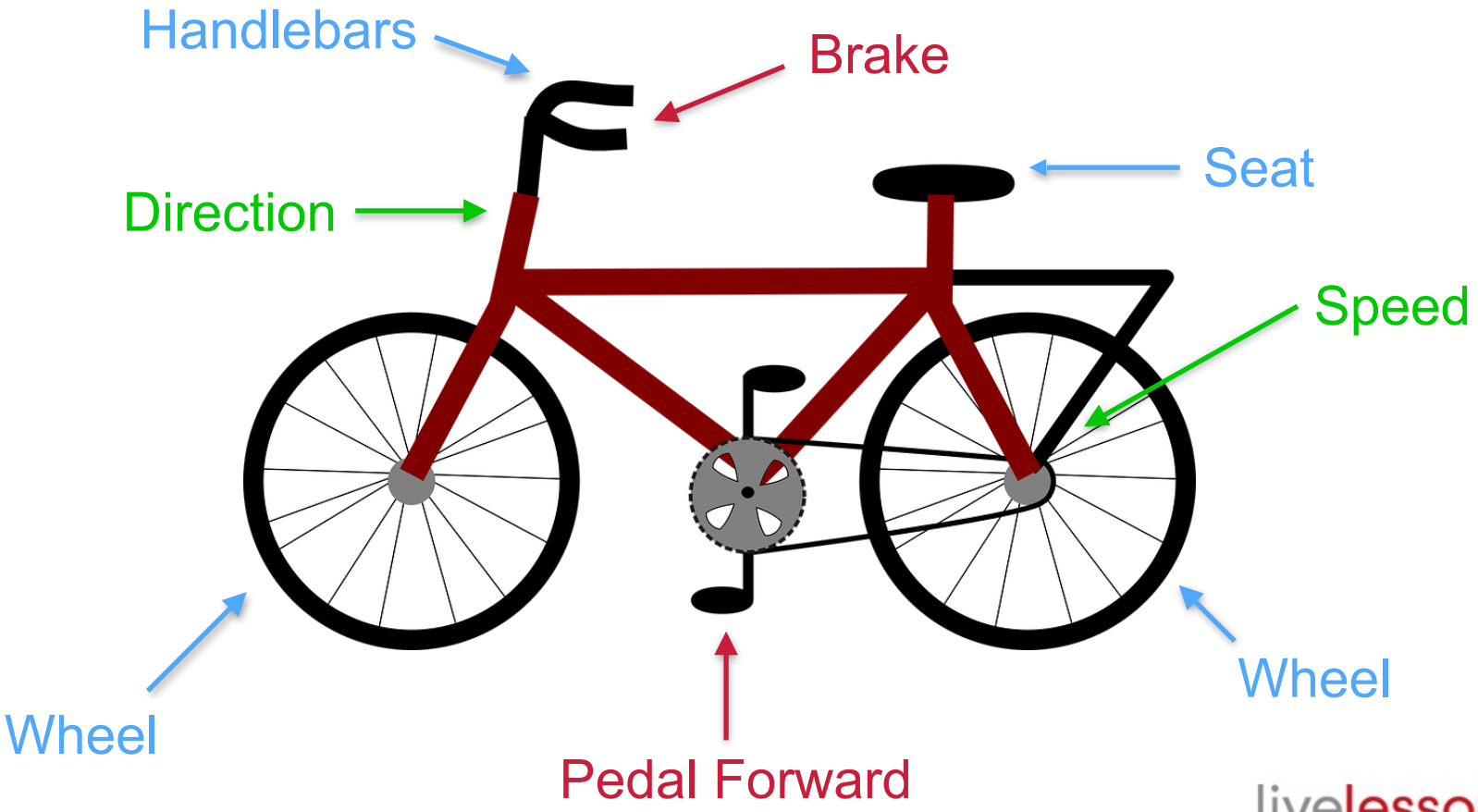
Attributes



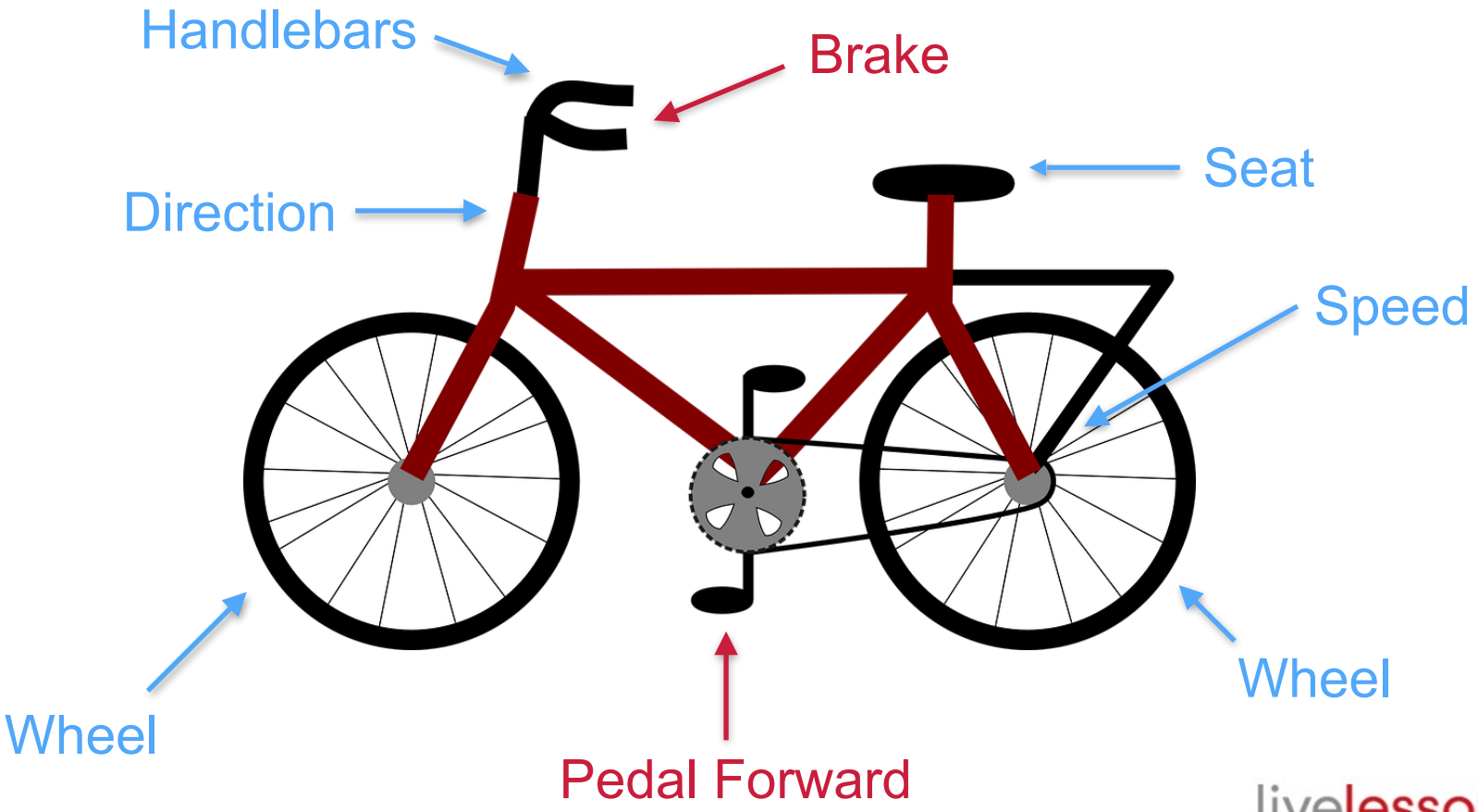
Methods



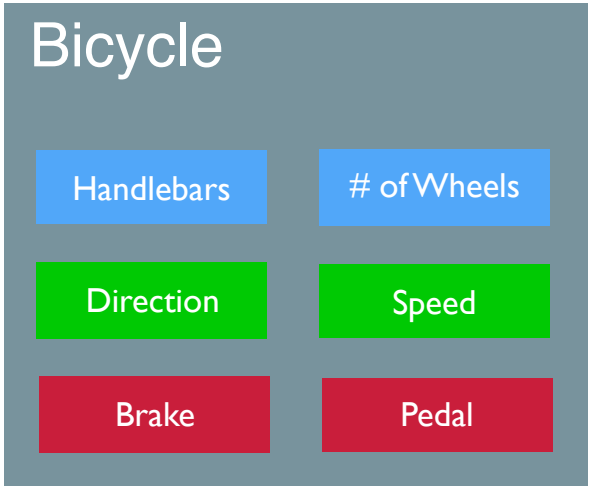
State



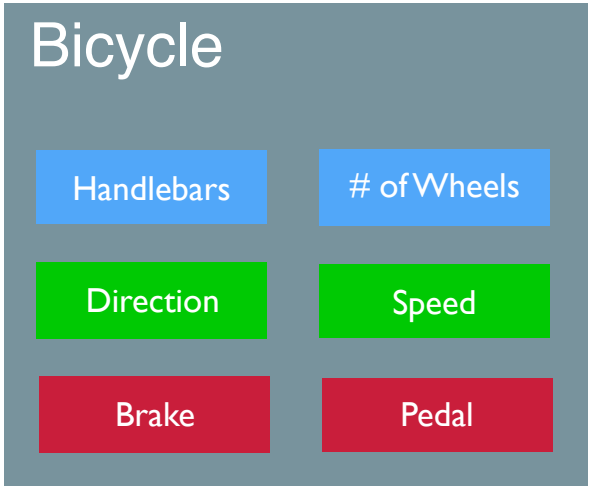
State (stored in Attributes)



Abstracted



Abstracted

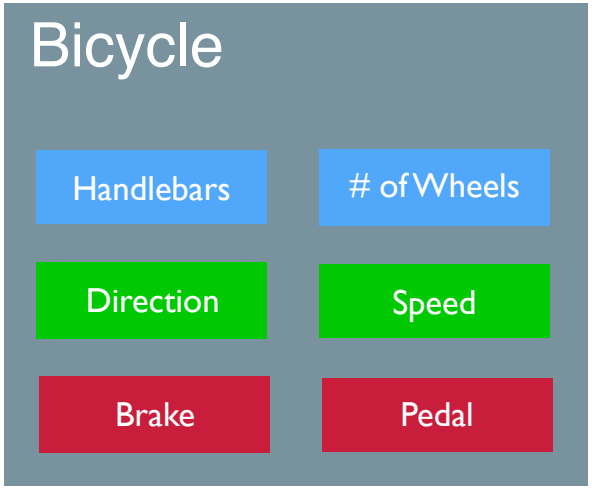


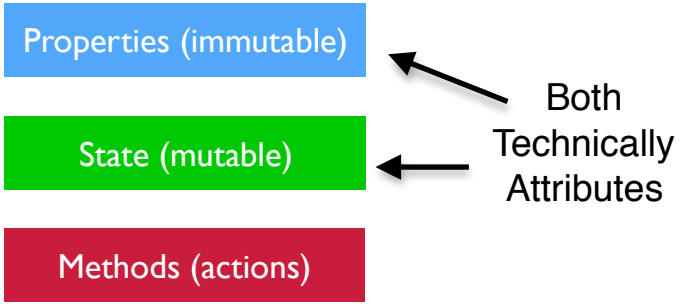
Properties (immutable)

State (mutable)

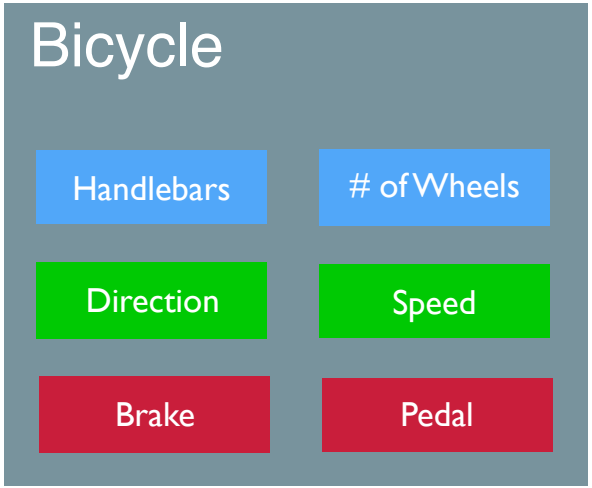
Methods (actions)

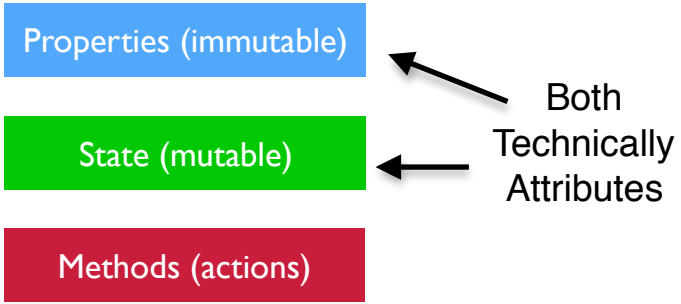
Abstracted



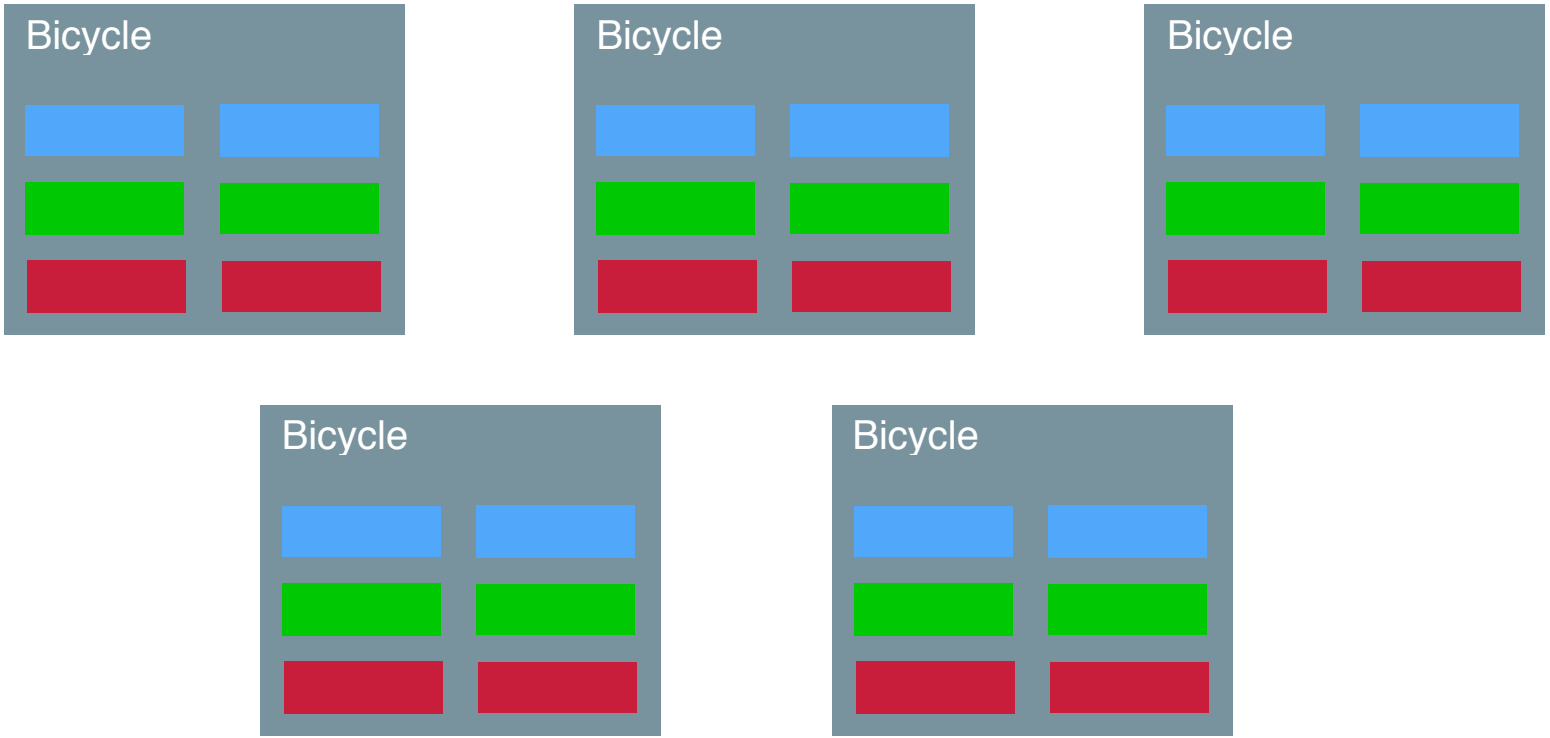


Class (think blueprint)





Objects and Instances



In Python… everything is

(secretly) an object!

Benefits

Lets talk about when we might benefit from OOP principles by applying them to our Foursquare Data

Benefits

Makes some things easier to program

Benefits

Makes other things harder to program

Benefits

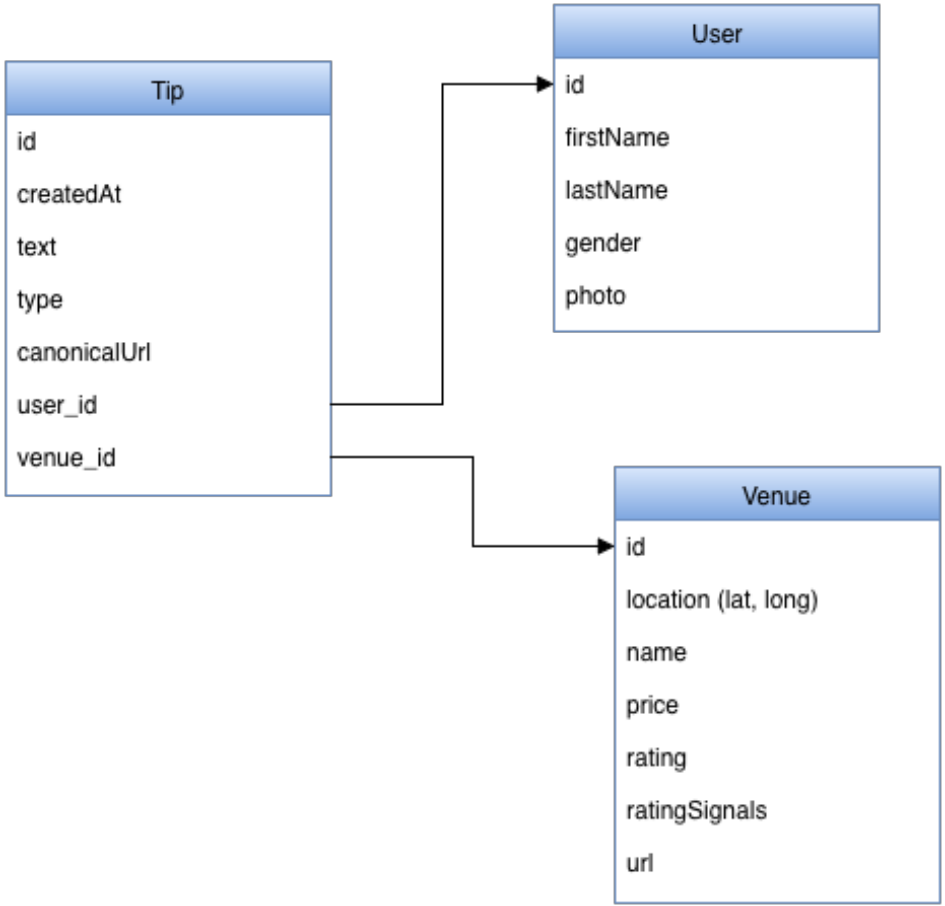
OOP allows you to combine data and functionality

Benefits

OOP allows you to combine attributes and methods

How might we represent our Foursquare data as objects?

Schema



Keyword Arguments

```
class Venue:
    def __init__(self, **data):
        self.id = data['id']
        self.name = data['name']
        self.price = data['price']
        self.rating = data['rating']
        self.ratingSignals = data['ratingSignals']
        self.url = data['url']
        self.latitude = data['latitude']
        self.longitude = data['longitude']
```

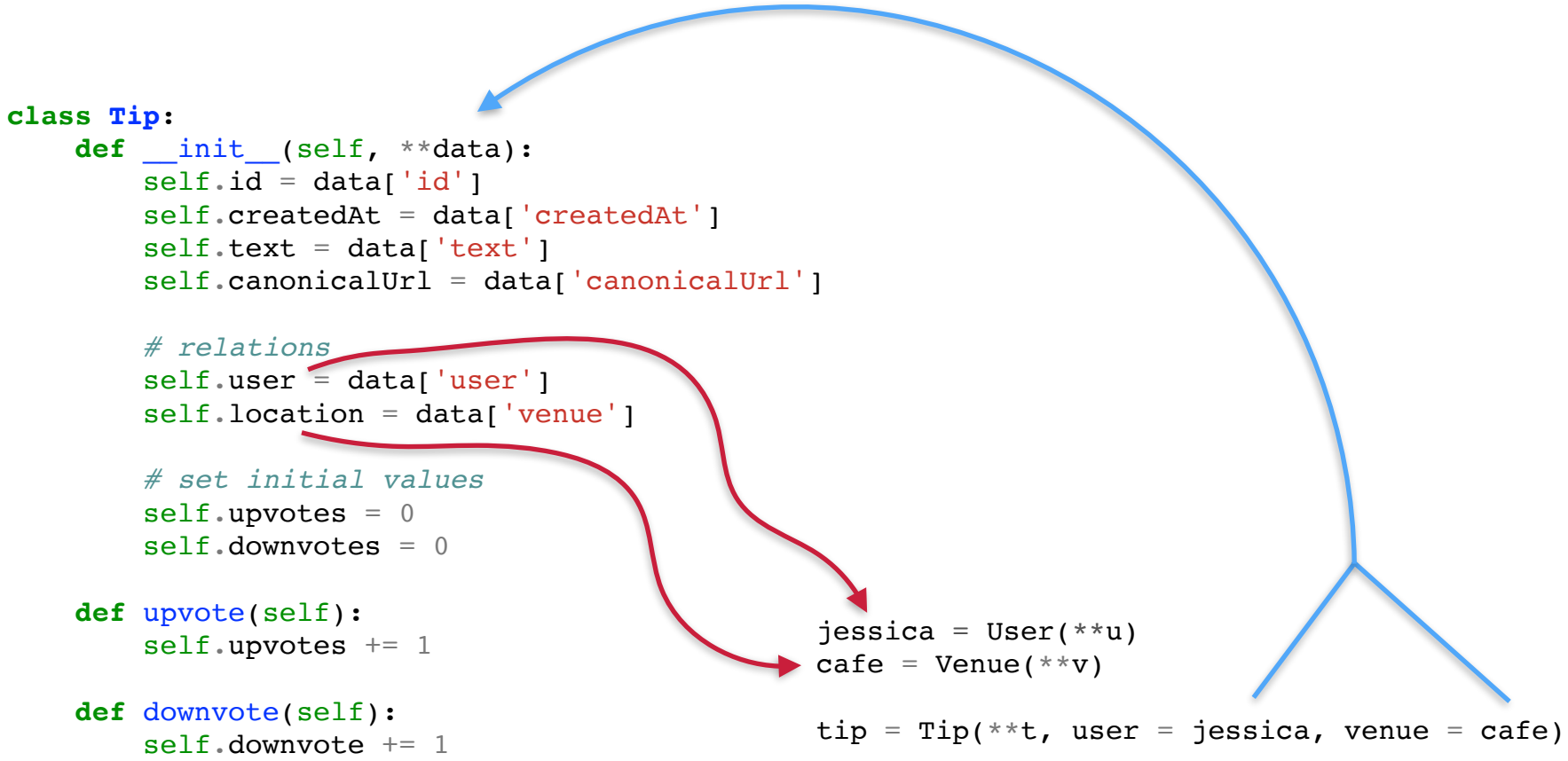
```
class User:
    def __init__(self, **data):
        self.id = data['id']
        self.firstName = data['firstName']
        self.lastName = data['lastName']
        self.gender = data['gender']
        self.photo = data['photo']
```

```
class Tip:
    def __init__(self, **data):
        self.id = data['id']
        self.createdAt = data['createdAt']
        self.text = data['text']
        self.canonicalUrl = data['canonicalUrl']
```

Benefits

Lets talk about when we might benefit from OOP principles by applying them to our Foursquare Data

Live Coding (in sublime): Just Data, lets add methods (fsmodels.py)



Inheritance

```
class Tip:
    def __init__(self, **data):
        self.id = data['id']
        self.createdAt = data['createdAt']
        self.text = data['text']
        self.canonicalUrl = data['canonicalUrl']

        # relations
        self.user = data['user']
        self.location = data['venue']

        # set initial values
        self.upvotes = 0
        self.downvotes = 0

    def upvote(self):
        self.upvotes += 1

    def downvote(self):
        self.downvote += 1
```

Inheritance

```
class Checkin:
    def __init__(self, **data):
        self.createdAt = time.time()

        # relations
        self.user = data['user']
        self.location = data['venue']
```

Inheritance

```
class Tip:
    def __init__(self, **data):
        self.id = data['id']
        self.createdAt = data['createdAt']
        self.text = data['text']
        self.canonicalUrl = data['canonicalUrl']

        # relations
        self.user = data['user']
        self.location = data['venue']

        # set initial values
        self.upvotes = 0
        self.downvotes = 0

    def upvote(self):
        self.upvotes += 1

    def downvote(self):
        self.downvote += 1
```

```
class Checkin:
    def __init__(self, **data):
        self.createdAt = time.time()

        # relations
        self.user = data['user']
        self.location = data['venue']
```

Inheritance

```
class Tip:
    def __init__(self, **data):
        self.id = data['id']
        self.createdAt = data['createdAt']
        self.text = data['text']
        self.canonicalUrl = data['canonicalUrl']

        # relations
        self.user = data['user']
        self.location = data['venue']

        # set initial values
        self.upvotes = 0
        self.downvotes = 0

    def upvote(self):
        self.upvotes += 1

    def downvote(self):
        self.downvote += 1
```

```
class Checkin:
    def __init__(self, **data):
        self.createdAt = time.time()

        # relations
        self.user = data['user']
        self.location = data['venue']
```

Benefits

OOP allows you to group/collect/arrange code in extensible patterns that reduce duplication

Benefits

And less code means less chances for error

(usually)

Use Cases

- OO Abstractions Naturally Fit Domain Abstractions (i.e. Video Games and GUI)
- Benefits of OOP Suit Problem Specifications (i.e. `scikit-learn`)

Use Cases

- OO Abstractions Naturally Fit Domain Abstractions (i.e. Video Games and GUI)
- **Benefits of OOP Suit Problem Specifications (i.e. `scikit-learn`)**

Benefits

How can we design a system/program that is consistent, interpretable, and extensible in the face of updating requirements?

Benefits

e.g. Designing a Machine Learning Library

Benefits

- Encapsulation and Modularization
- Inheritance and Extensibility (DRY)
- Composition and Separation of Concerns

When to Use

- Application Programming Interfaces
- Multiple Independent (interacting) Agents
- Concurrent Components (through message passing)
- Long Lived Stateful Objects

When to Use

- Data Structures, Libraries, Frameworks, etc.
- Simulations (and Games)
- Distributed Systems*
- Graphics

* Implementation Dependent (relies on asynchronous messaging)

When to Use

- Custom Modules and Packages
- Poker and Blackjack (i.e. expected returns of strategies)
- Computer Multitasking (and Operating System Interfaces)
- GUI windows

When to Use

- Custom Modules and Packages
- Poker and Blackjack (i.e. expected returns of strategies)
- Computer Multitasking (and Operating System Interfaces)
- GUI windows

When Not to Use

- Small Scripts and “Glue” code
- Anything Procedural (think recipes and algorithms)
- Anything Functional (can you write it as an equation?)
- Data flow and Stream Processing (data pipelines)

https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism

At the end of the day it is mainly personal preference

Remember… quick iteration is key to data science!

Avoid tight coupling and be careful with mutating data

And you can happily mix paradigms

(`scikit-learn` does)!

Object Oriented Programming

The big idea is "messaging" -- that is what the kernal of Smalltalk/ Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase)… The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

- Alan Kay

source: http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

Object Oriented Programming

Think of the internet -- to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas.

- Alan Kay

source: http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html