Overview of Urban Data Handling

Huy T. Vo

9/20/2015

Logistics

- eDAP program (check with the education team)
- Office hours are flexible — or an extra working session (please send us a note if you prefer a working session)
- Plan for today:
  - Handling urban data
  - ~~Plotting principals~~
  - Lab: handling urban data

Urban Data Sources

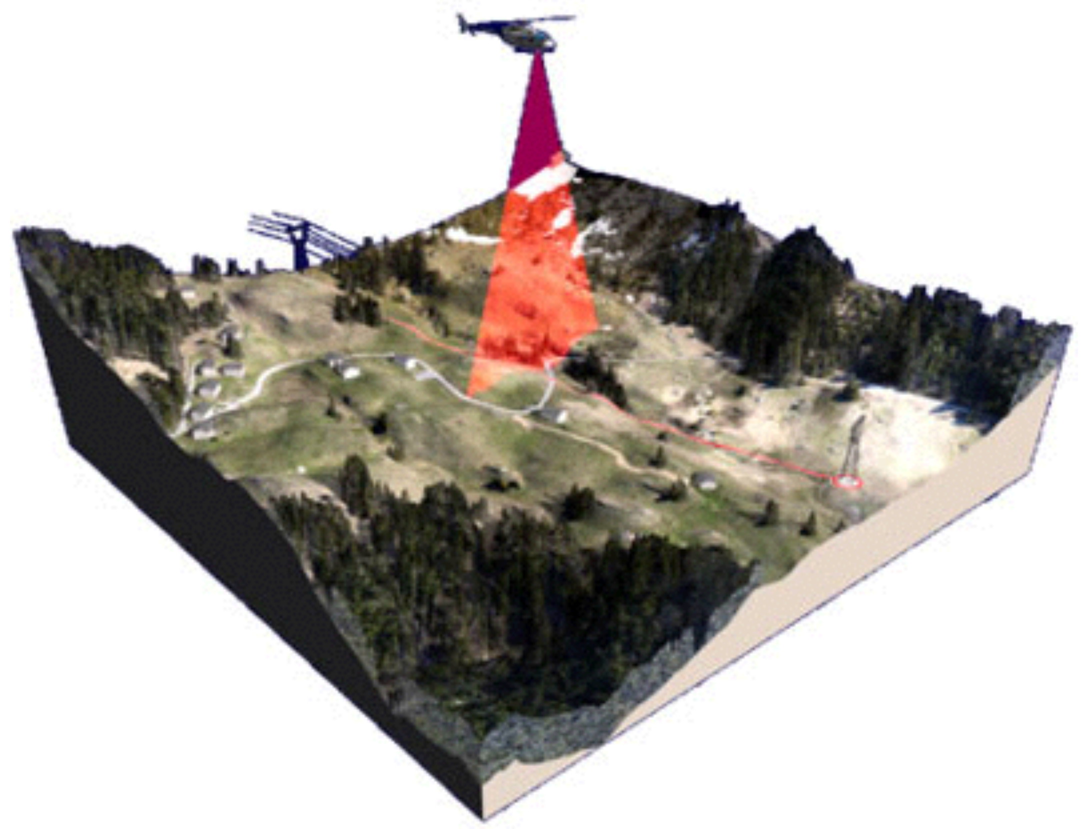
- **Organic data flows**
  - Administrative records & transactions (census, permits, sales…)
  - Operational (traffic, transit, utilities, health system, …)
  - News and social media (Twitter feeds, blog posts, Facebook, …)
- **Sensors**
  - Personal (location, activity, physiological)
  - Fixed *in situ* sensors
  - Crowd sourcing (mobile phones, …)
- **“Novel” sensor technologies**
  - Visible, infrared and spectral imagery, RADAR, LIDAR
  - Gravity and magnetic, seismic, acoustic
  - Ionizing radiation, biological, chemical

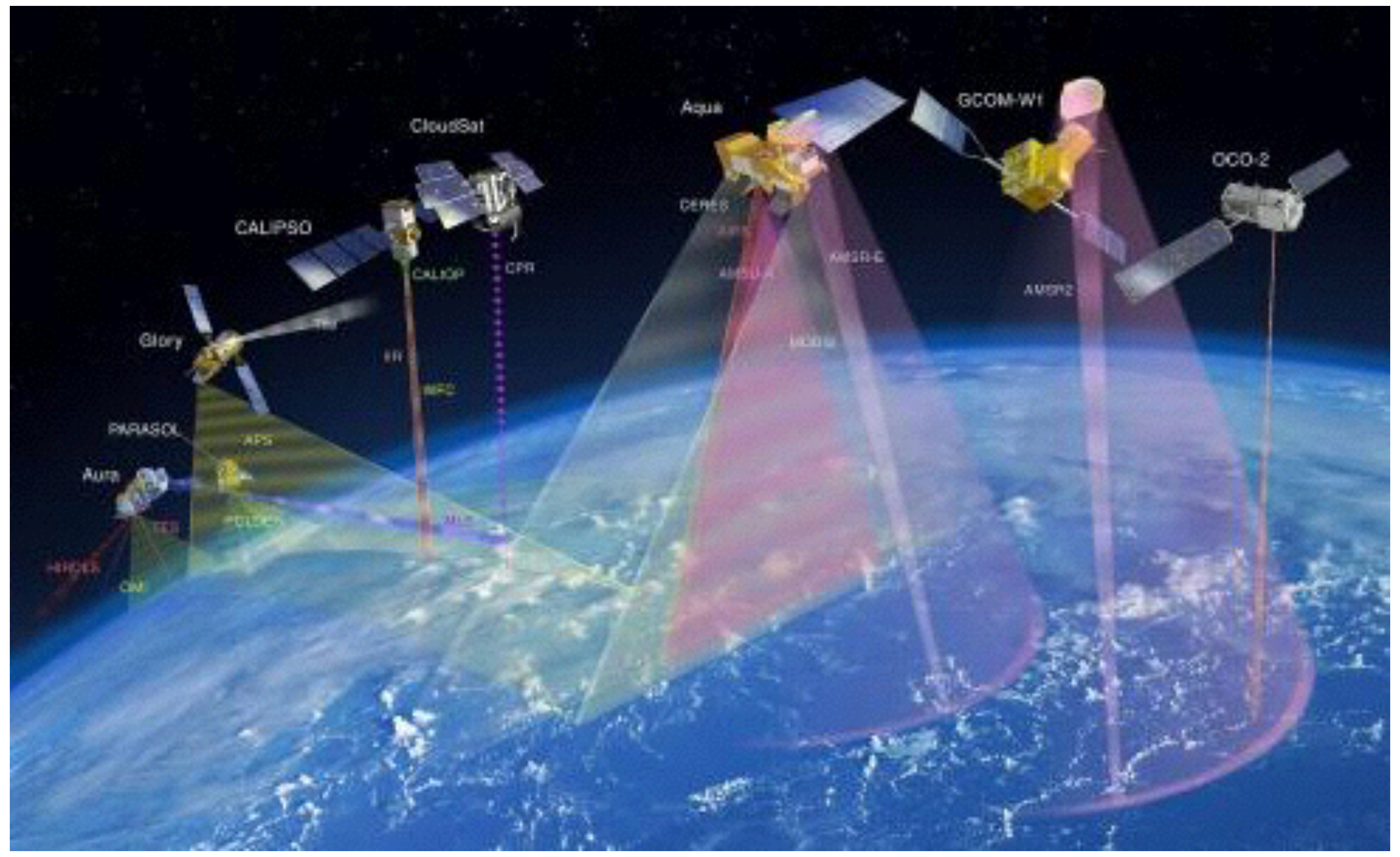
Record and Transactional Data

- Indivisible data unit describing an event or an object
  - A taxi trip, an accident report, a FourSquare check-in, or a 311 request
- Data are separable: each is *necessary* and *sufficient* to describe an event/object
- *Usually* has a fixed schema, i.e. the number of fields and types, to increase readability for human as well as machines
  - Tables: a list of records (aka rows) with a fixed number of fields
- Transactions may contain references to other data sources (by ids, names, etc.)
  - *Usually* represented by groups of tables in relational databases

Sensor Data

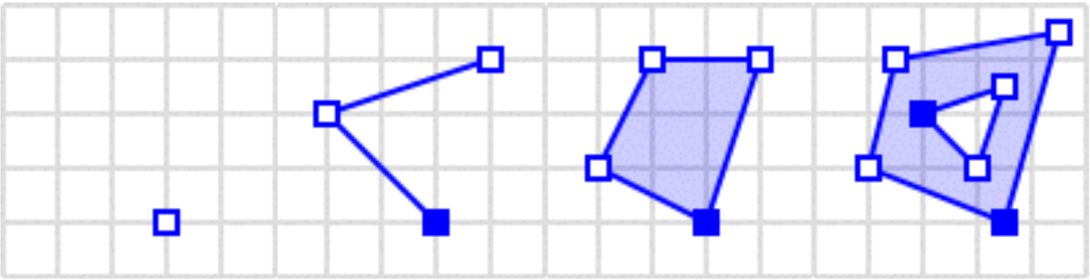
- Remote Sensing
  - Visible: active/passive
  - Thermal IR or Microwave
- In-situ Sensing
  - Sensors + SBC/Smart devices
  - Internet of Things
- Tracking things through space and time



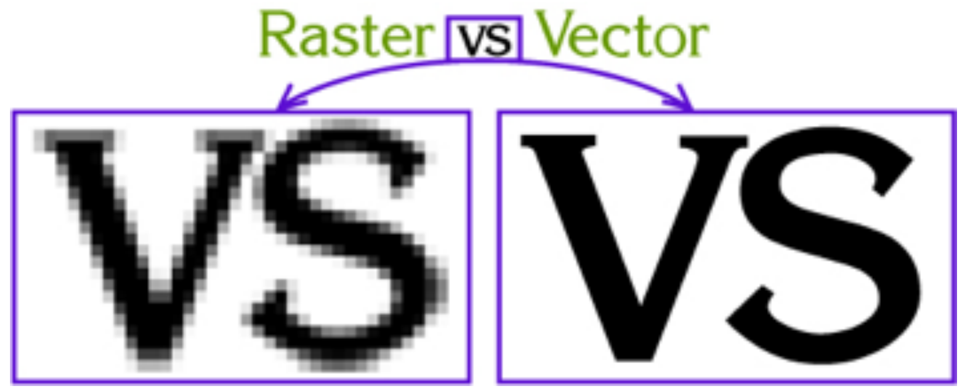


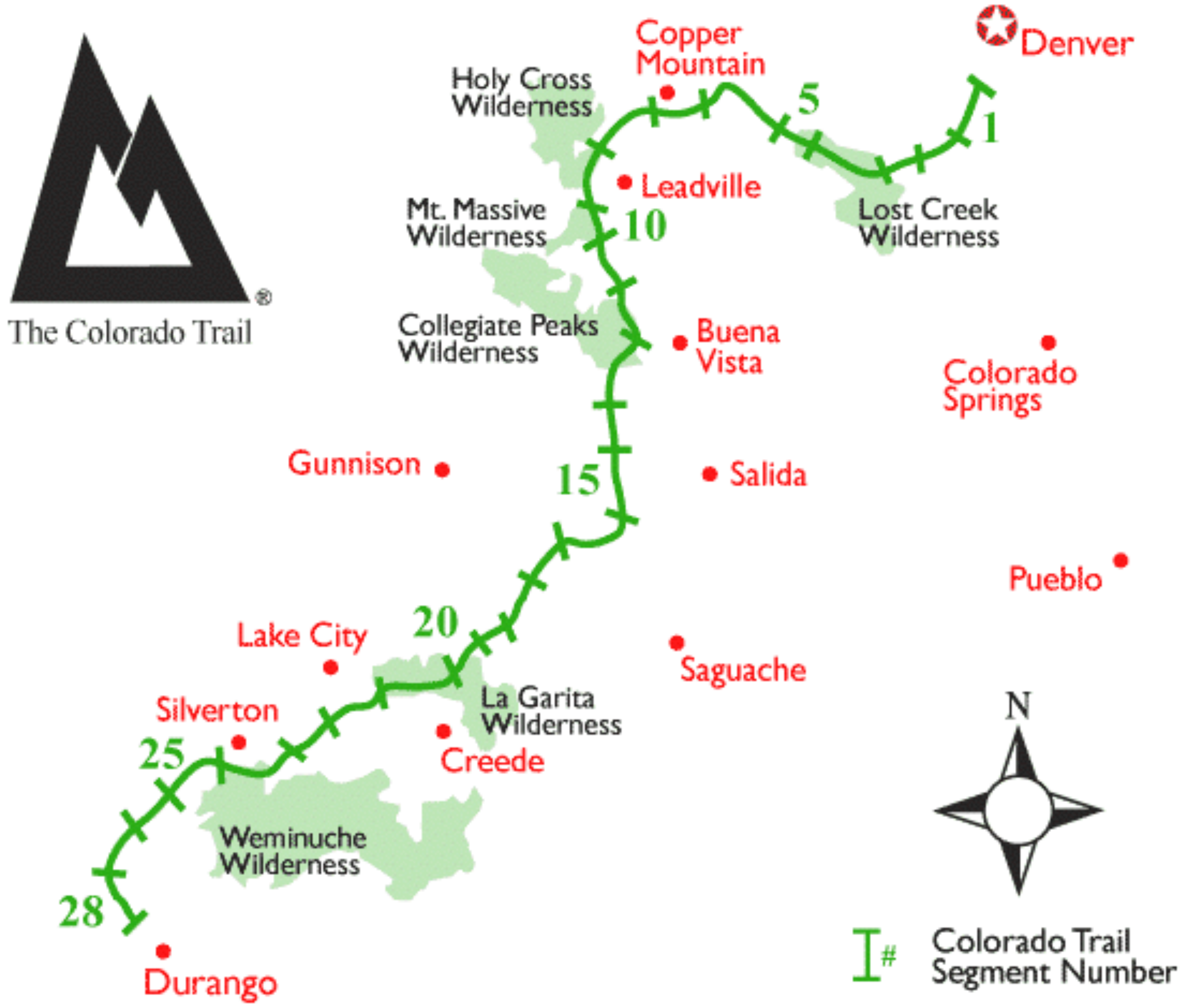
Geospatial (GIS) Data

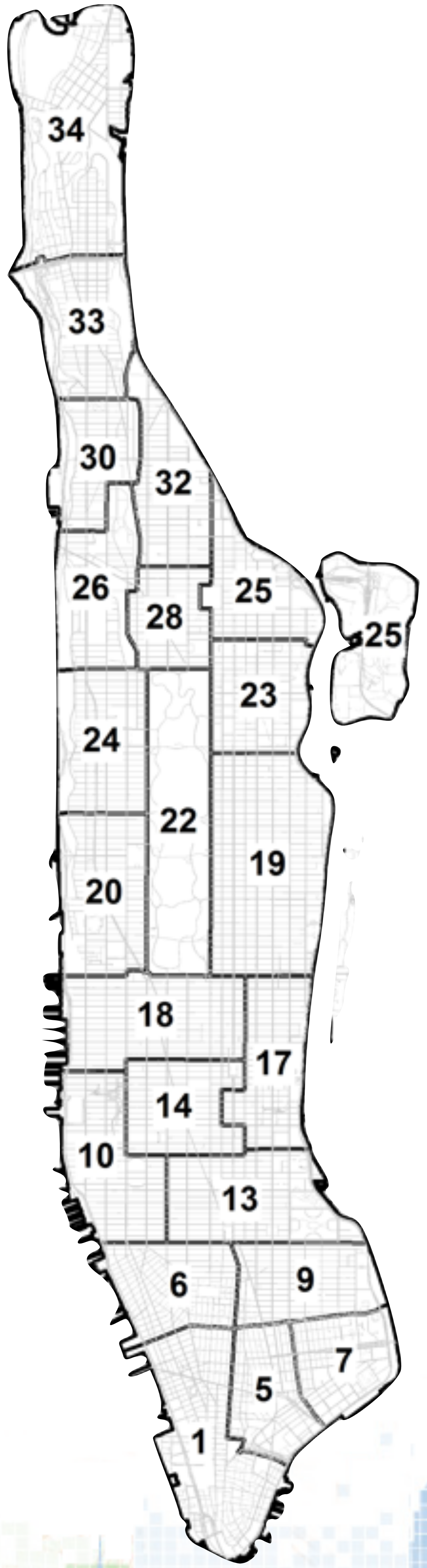
- Data with explicit geographic locations or features
- Vector data:



  - Points: GPS coordinates, point of interests
  - Lines: road networks, bike trails
  - Polygon: state boundaries, park areas
  - 3D Shapes: city models, terrain mapping
- Raster data: geo-referenced images
  - Satellite photos, overlay visualizations



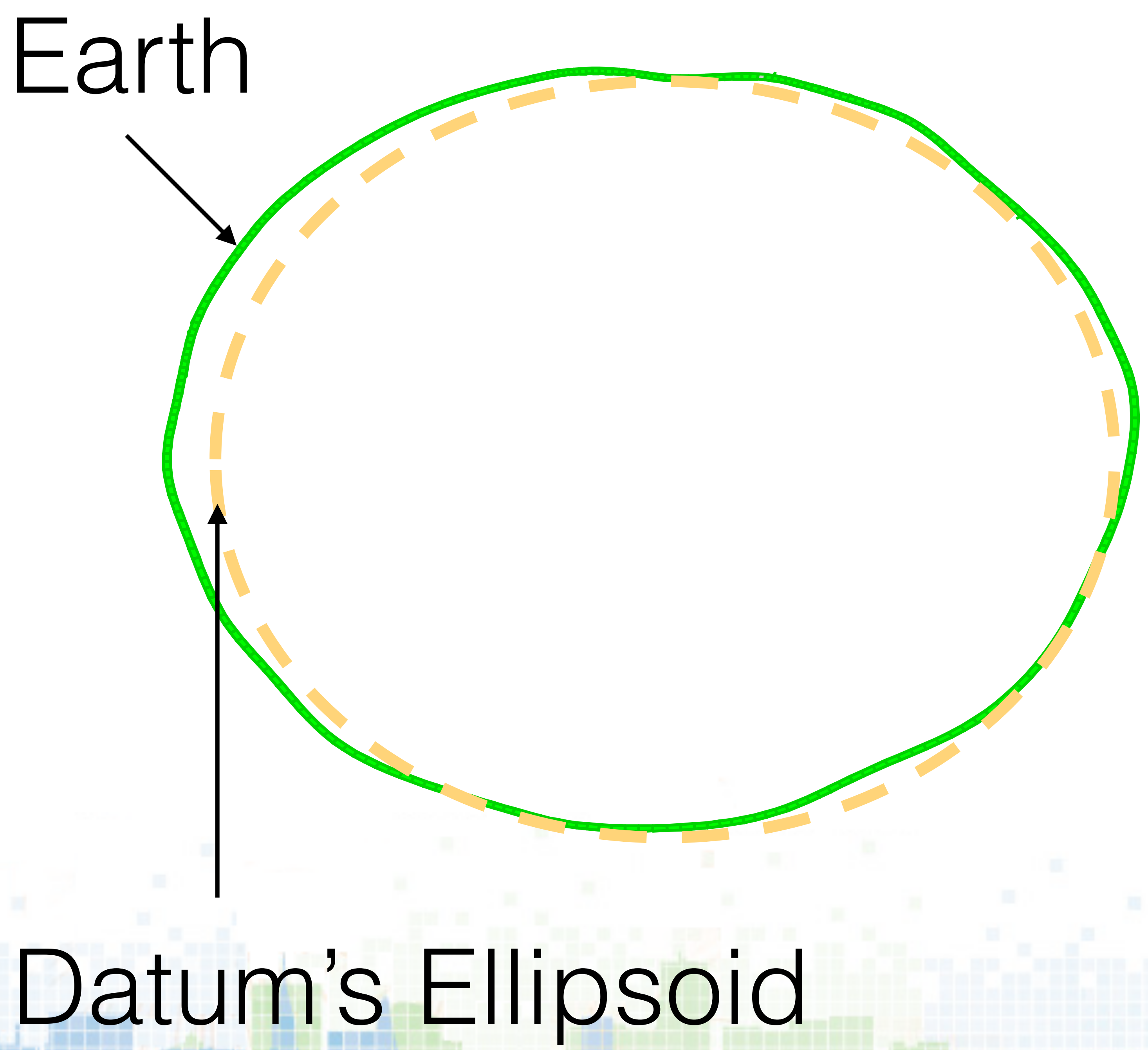


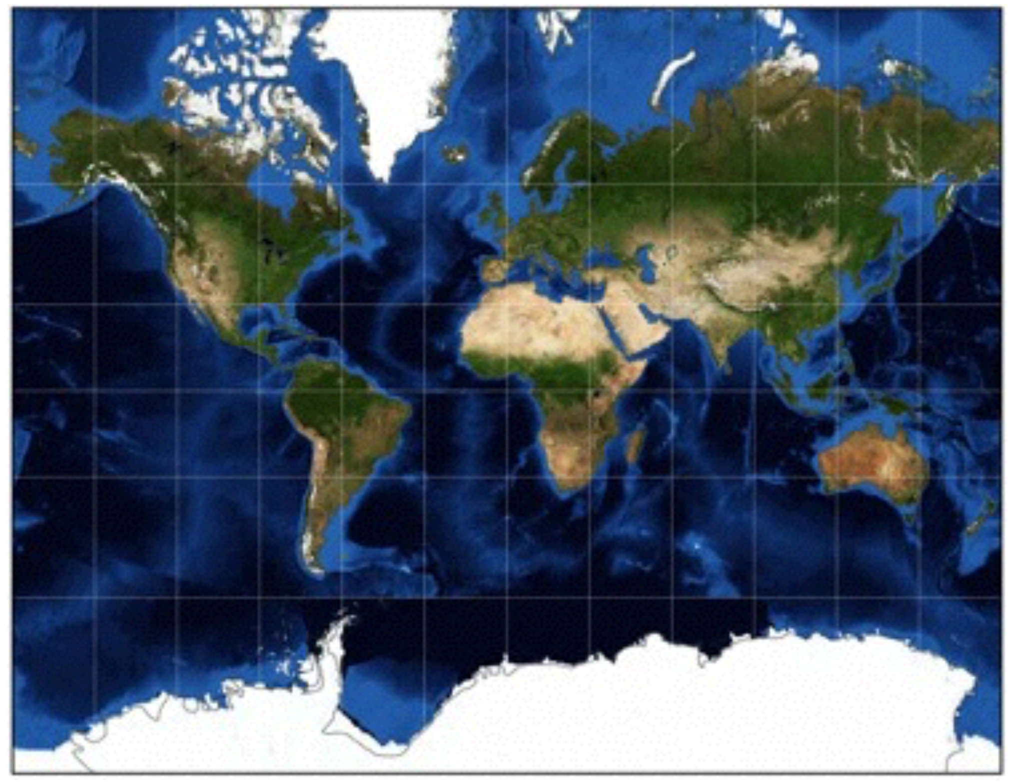


[Images © Wikipedia, runtrails.net, landsurveyorsunited.com, The Office of the Manhattan Borough President]

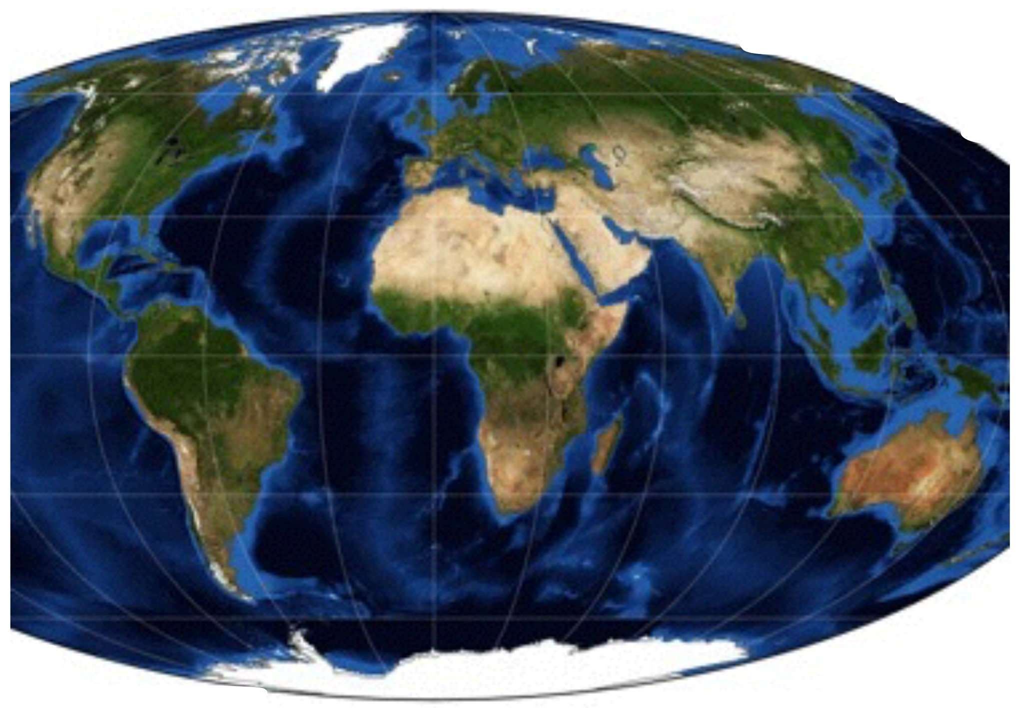
Geographic Coordinate System: Datum and Projection

- Datum: how the earth is being modeled (as an ellipsoid)
- Projection: flattening the 3D ellipsoid onto a 2D surface





Mercator



Mollweide

Azimuthal

[Images © MapToasterTopo]

Metadata

- “data about data”: annotate, label and describe characteristics of a data set for easier retrieval and interpretation of the data
  - Data dictionary/schema (types and structures), map projection!
  - Data profiles and summary statistics (value ranges, missing values, etc.)
  - Data provenance including the data ingestion process for auditing and reproducibility purposes
  - Usage statistics: inferring data value

Urban Data — Space + Time

- Spatio-temporal Exploration: where + when [+ what]
  - Which areas are known to have power outage in major storm events?
  - When did the last burglary occur anywhere around a neighborhood?
  - What types of business open within a year of new residential buildings?
- Spatio-temporal Analysis: trends, anomalies and prediction
  - Moving vehicles — studying patterns of mobility
  - Land use over time — predicting long-term usage and development

Temporal Representation

- A single string:
  - `"2/1/2015 13:05"`
  - `“Sun 02/01/2015 13:05:30 GMT"`
  - `"2015-02-01`**`T`**`13:05:30`**`Z`**`"` (ISO 8601)
- Separate date and time: `("2/1/2015", "13:05:30")`
- Separate components: `(2015,2,1,13,05,30)`
- A timestamp: `1422813930` (Unix/epoch time — seconds since 1/1/1970)
- Format: usually stored as strings or numbers

Handling Temporal Data — Python

```
>>> from datetime import datetime
>>> datetime.fromtimestamp(1422813930)
    datetime.datetime(2015, 2, 1, 13, 5, 30)
>>> datetime.fromtimestamp(1422813930).isoformat()
    '2015-02-01T13:05:30'
>>> datetime.fromtimestamp(1422813930).isoformat(' ')
    '2015-02-01 13:05:30'
>>> datetime.fromtimestamp(1422813930).ctime()
    'Sun Feb  1 13:05:30 2015'
>>> d = datetime.strptime('2015-02-01 13:05:30', '%Y-%m-%d %H:%M:%S')
>>> d.isocalendar()
    (2015, 2, 1)
>>> import time
>>> time.mktime(d.timetuple())
    1422813930.0
```

Spatial Representation

- Vector geometry: sequences of coordinates — e.g. latitude and longitude
  - Point: `(40.7127,-74.0059)` or `(583968.1,4507339.1,10)`
  - Lines/Polygon: `[(40.7127,-74.0059), (40.8127,-74.0059)]`
  - Geometric operations (point in polygon test, intersection test, etc.) are performed on primitives
- Raster geometries: image + metadata (e.g. enclosed geographic bounds)
  - Geometric operations are performed on pixels/fragments
- Location information: postal address, place name, BBL, zipcode, etc.
  - Often get geocoded (convert geographic coordinates) before processing

Spatial Format — Plain text

- Best for embedding in documents, human readable
  - WKT: well-known text, purely geometries, lots of support from well-known DBs

```
POINT (-74.0059 40.7127)

LINESTRING (-74.0059 40.7127,-74.0059 40.8127)
```

  - GeoJSON: support attributes, web app friendly (there're also TopoJSON/KML)

```
{"type": "Feature",
 "geometry": {
  "type": "Point",
  "coordinates": [-74.0059,40.7127]
 },
 "properties": {"name": “My Point"}}
```

  - SVG: scalable vector graphics — focus on rendering (not geographic purely)

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4" stroke="pink" />
  <circle cx="125" cy="125" r="75" fill="orange" />
  <polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-width="4" fill="none" />
</svg>
```

Spatial Format — Binary

- Best for efficient data manipulation (e.g. in databases)
  - Shapefile: from ESRI ArcGIS, a collection of files describing primitive records (.shp) , their indices (.shx) and attributes (.dbf)
    - Widely used in the GIS community (almost a “standard”)
  - WKB: well-known binary, a binary version of WKT, lots of DB support
- Most raster data are in binary
  - GeoTIFF: a TIFF image + georeferencing information
  - JPEG2000: Geography Markup Language (GML) for georeferencing

Handling Spatial Data — Python

- Most data can be stored in Python native structures: tuple, list, and dictionaries
- Useful packages for handling spatial data:
  - **`fiona`** : read/write GIS files (a thin API for the GDAL/OGR I/O library)
  - **`geopandas`** : extending **`pandas`** with geo support
  - **`json`** : a built-in package for reading JSON files including GeoJSON
  - **`pyproj`**: map projection (conversion from one projection to another)
  - **`shapefile`** : a light-weight package for reading shapefiles
  - **`shapely`** : provide geometric operations on 2D planes (oblivious to geographic projections), based on PostGIS engine GEOS

Handling Spatial Data — Conversion

GDAL/OGR provides a powerful command line tool for conversion (and transformation) of geospatial data, similar to ImageMagick's convert: **`ogr2ogr`**

https://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries

- Convert shapefile to geojson

```
ogr2ogr -f GeoJSON nyc.geojson nyc.shp
```

- … also reproject the data to EPSG:4326 coordinates (~GPS lat lon):

```
ogr2ogr -f GeoJSON -t_srs EPSG:4326 nyc.geojson nyc.shp
```

Handling Spatial Data — Projection

```
>>> import pyproj

## Create the MapPLUTO projection EPSG:2263
>>> proj = pyproj.Proj(init='EPSG:2263', preserve_units=True)

>>> nycLatitude  = 40.692547
>>> nycLongitude = -73.928113

## Map the NYC lon/lat coordinates onto MapPLUTO
>>> proj(nycLongitude, nycLatitude)
    (1004185.1129963363, 191598.17059893996)

## Map a MapPLUTO coordinates back to lon/lat
>>> proj(1004185.1129963363, 191598.17059893996, inverse=True)
    (-73.928113, 40.69254699999981)
```

Tabular Data Representation

- Flat structured data with a fixed schema:
  - Every record (row) has the same set of variables (columns)
- The “standard” way to manipulate and share record data
  - Excel/Google Spreadsheet are virtually available everywhere
- Map well to (relational) databases
- Data are usually stored in textual format
  - Spatial and temporal information can be embedded as cell values

Tabular Data Format

- Excel spreadsheet (.XLS, .XLSX)
  - Can be open by many applications (LibreOffice, Apple's Numbers,…)
  - But still native to Excel — advanced features (plotting, complex function, customized filters, etc.) are only supported in Excel
- LibreOffice or Apple's Numbers: similar to Excel, certain features are only available in their native formats. Be cautious when sharing data!
- Google Spreadsheet: accessible online but with limited functionalities
- All depends on a spreadsheet editor to manipulate and analyze data
  - How to process them in Python? How to share data across platforms?

Comma-Separated Value (CSV file)

- Data are stored in plain-text: readable by both human and machines
- Records/rows are separated by lines
- Fields are separated by commas (,)

```
start time,station name,station latitude,station longitude
2/1/2015 0:00,8 Ave & W 31 St,40.75044999,-73.99481051
```

- Commas can be included in data fields by using double quote "

```
start time,station name,station latitude,station longitude
2/1/2015 0:00,"8 Ave , W 31 St",40.75044999,-73.99481051
```

Use the Python's csv package to handle the quoted values correctly!

Delimiter-Separated Values

- Like CSV but may use other characters as delimiter instead of comma
  - Most CSV are treated as delimiter-separated values. Python's `csv` package support read/write "DSV" files!
- Tab-Separated Values are TSV
- Simple guidelines:
  - Real comma-separated : .CSV
  - Tab-separated: .TSV
  - Other delimiter separated: .TXT (e.g. Excel will detect/ask for delimiters)

Limitations of CSV/Tabular Format

- Not suitable for hierarchical data structures
  - Grouping of data by city, county, census tract, etc.
- Only deal with static schemas:
  - Cannot add a new field (e.g. notes) for some records without creating an entire column
- Example: NYC 311 Service Request data set has 53 columns
  - Most of which are unspecified, depending on the service type
  - But still taking space both in term of storage and on-screen visualization

NYPD Motor Vehicle Collisions — Table

- Tabular Representation

Date	Vehicle Type 1	Vehicle Type 2	Vehicle Type 3	Vehicle Type 4	Vehicle Type 5	…
9/12/2015	Taxi	Van	unknown			
9/10/2015	Passenger Car	unspecified	unspecified	unspecified	unspecified	

- Always have 5 fields (some are unknown/unspecified/NA)
  - Harder to detect the actual number of vehicles involved
  - "Vehicle Type 1" and "Vehicle Type 2" are technically two distinct entities (cannot sort or filter accidents involved taxi as a vehicle)

Semi-structured Data Representation

- Appropriate for data that has irregular structure
  - Some elements only exists in certain records
  - The number of elements are varied
  - Flexible to shared documents among systems and databases: has fixed grammar (for data specification) but not data schema
  - Easier to parse by machines
- Popular representations: tree-structured (XML) and key-value stores (JSON) — good fit for *NoSQL* databases (later in the semester)

The Extensible Markup Language (XML)

- XML defines a set of syntax for describing data and/or data specifications
- But it can be used to describe pretty much anything
  - Can include image as encoded binary string
- Elements are encapsulated by tags in a nested hierarchy structure
  - An element may have children — aka containing a list of items
- XML collections can be queried through XQuery : a functional query language for XML
  - This is where data are being treated as stricter types (dates, numbers, etc.)

NYPD Motor Vehicle Collisions — XML


```
<XML>
    <Name>NYPD Motor Vehicle Collisions</Name>
    <Collision>
        <Date>9/12/2015</Date>
        <VehicleType>Taxi</VehicleType>
        <VehicleType>Van</VehicleType>
        <VehicleType>unknown</VehicleType>
    </Collision>
    <Collision>
        <Date>9/10/2015</Date>
        <VehicleType>Passenger Car</VehicleType>
    </Collision>
</XML>
```

JavaScript Object Notation (JSON)

JSON is designed for mapping data onto computer language data structures (derived from JavaScript), to avoid the overhead of parsing data received on web browsers.

- Support native JavaScript types key/value pairs (dictionary), list/tuple, string, numbers, etc.
- Open standard, simple enough to be implemented in many languages including Python (the `json` package).
- Lots of supports from the web-based community
- Human-readable and works very well with Python's data structure.

NYPD Motor Vehicle Collisions — JSON

```
{
    "Name": "NYPD Motor Vehicle Collision",
    "Collisions": [
        {
            "Date": "9/12/2015",
            "VehicleType": ["Taxi", "Van", "unknown"]
        },
        {
            "Date": "9/10/2015",
            "VehicleType": ["Passenger Car"]
        }
    ]
}
```

In a nutshell, it's a Python dictionary…

JSON in Python

```
>>> import json

>>> inputFile = open("file.json", "r")

>>> data = json.load(inputFile)

>>> print data
  {'Collisions': [{'Date': '9/12/2015', 'VehicleType': ['Taxi', 'Van',
  'unknown']}, {'Date': '9/10/2015', 'VehicleType': ['Passenger
  Car']}], 'Name': 'NYPD Motor Vehicle Collision'}

>>> print data['Collisions'][0]['VehicleType']
    ['Taxi', 'Van', 'unknown']
```

How to retrieve data?

- Data are often obtained through a variety of mechanism:
  - Offline (Fedex portable hard-drive, USB sticks, etc.)
  - Ad-hoc data sharing (email, download links)
  - Shared space on the cloud (DropBox, Google's Drive, etc.)
  - Web scraping — getting all listings from a Yelp search page
  - Online — data portals and an API
    - Avoid the need for repetitive data requests, suitable for machine processing or batch jobs

Online Data Access API

- Request are usually self-contained within an URL template (REST API):

  - Requesting locations of all B52 bus using your developer keys

    http://api.prod.obanyc.com/api/siri/vehicle-monitoring.json?key=API_KEY&VehicleMonitoringDetailLevel=calls&LineRef=B52

  - Returns a list of followers for a specified user:

    https://api.twitter.com/1.1/followers/list.json?screen_name=USERNAME

most are in JSON!

- Static feeds — where data are updated periodically (or in real-time):

  - Status of all CitiBike's station: https://www.citibikenyc.com/stations/json

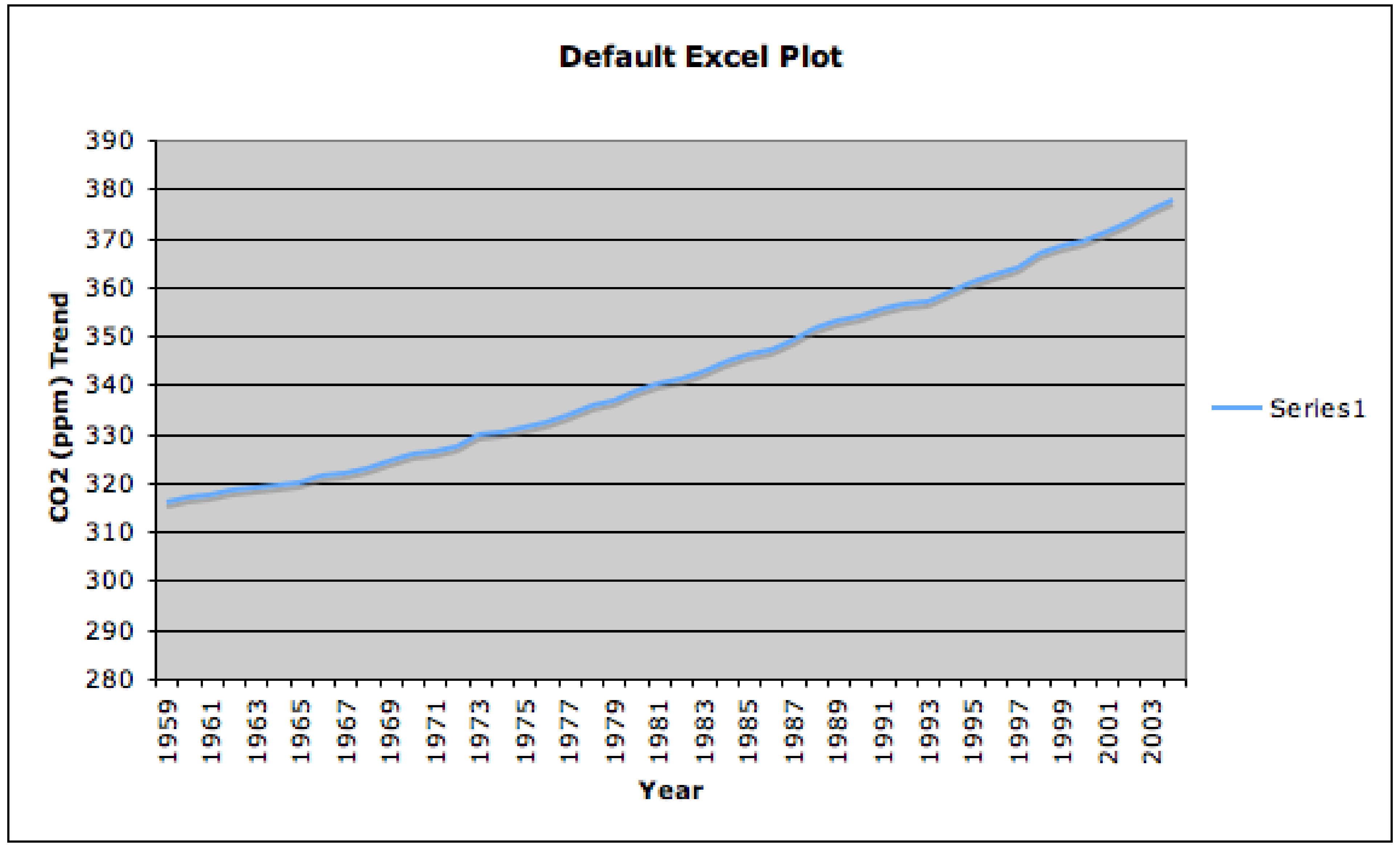
  - MTA Service Status: http://web.mta.info/status/serviceStatus.txt (XML)

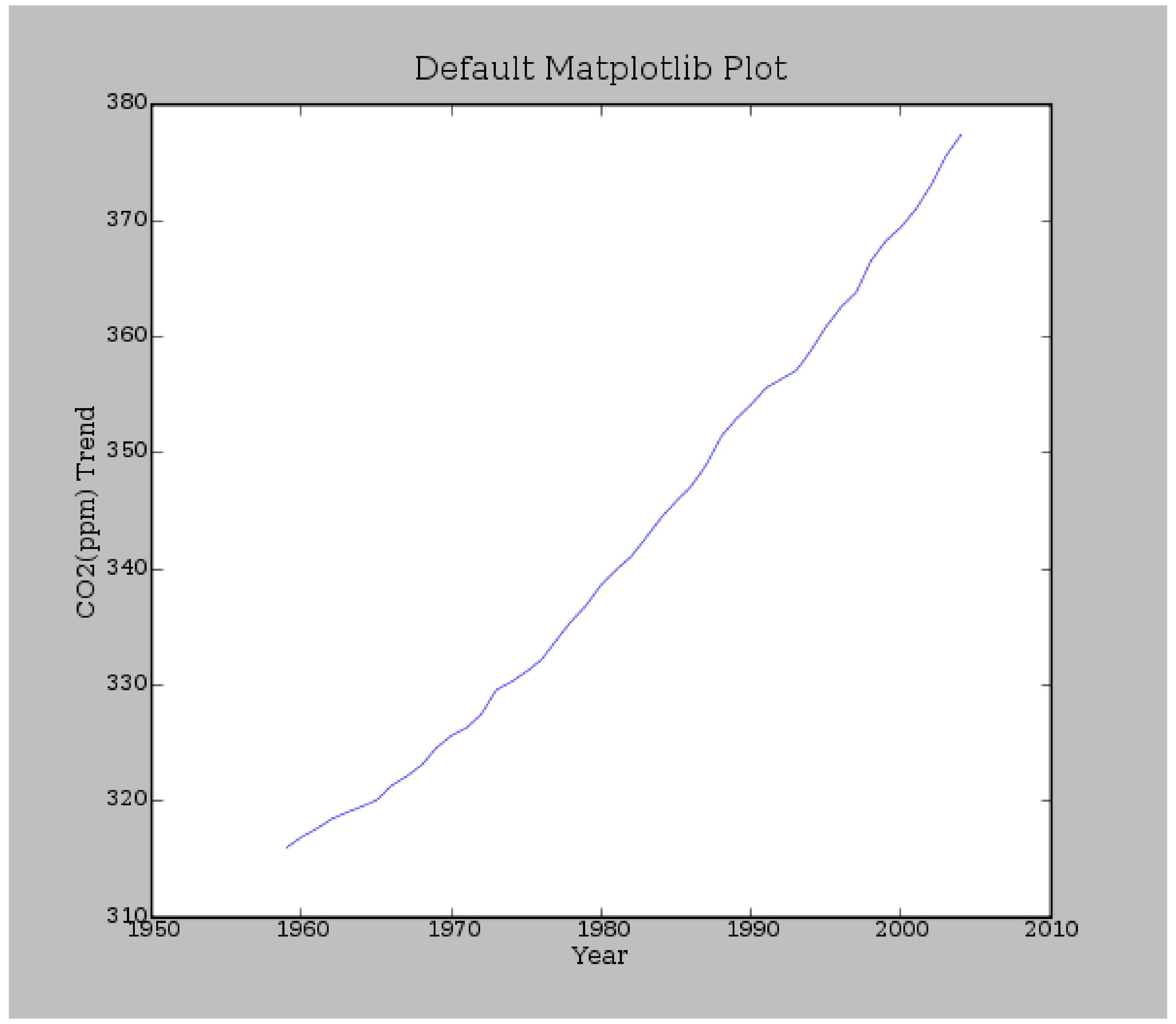
Fetching data online with Python

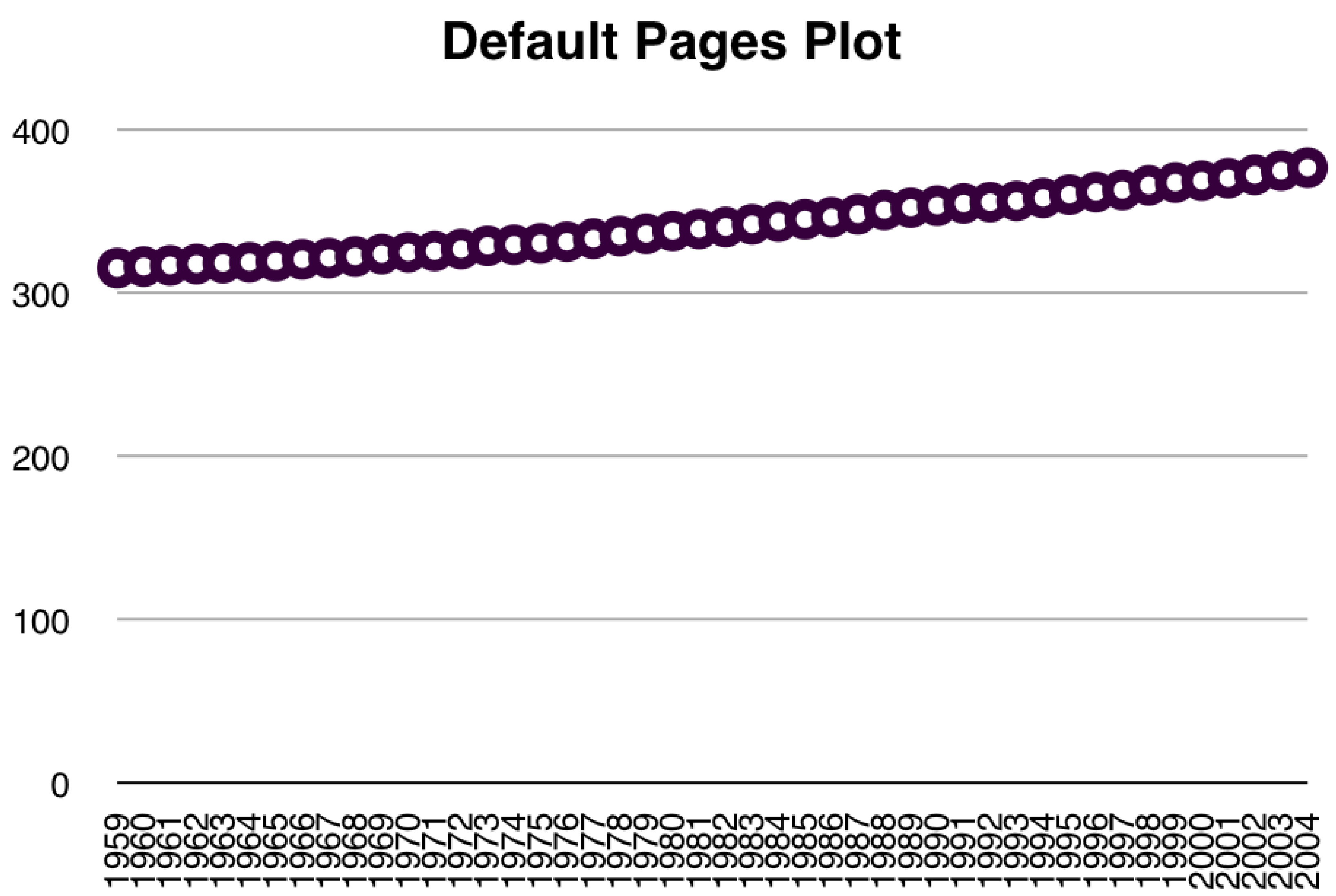
```
>>> import json
>>> import urllib2

>>> url = 'https://www.citibikenyc.com/stations/json'
>>> request = urllib2.urlopen(url)
>>> data = json.loads(request.read())

>>> print len(data['stationBeanList'])
  508

>>> print data['stationBeanList'][0]['availableDocks']
  36
```

Until next time…

- What's wrong with these plots?







[Examples from Callahan et. al]

Thank you!