```
[1]: from matplotlib.pyplot import *
     %matplotlib inline
     from IPython.display import Image

     import warnings
     warnings.filterwarnings('ignore')
```

# DS1 Lecture 06

**Jim Bagrow**

**Last time:**

1. Social ranking and uncertainty - Lower confidence bound (LCB) sort

**Today's plan:**

1. Data science "pipeline"
2. Typology of data
3. Storing data
    - Text files (CSV), JSON, XML, databases
    - Delimiter collisions
    - How to choose the right format?

(Please be sure to read any parts we may have skimmed over!)

Of course, storing and retrieving data are connected: once you've retrieved some data, you will want to store it!

## Data Science

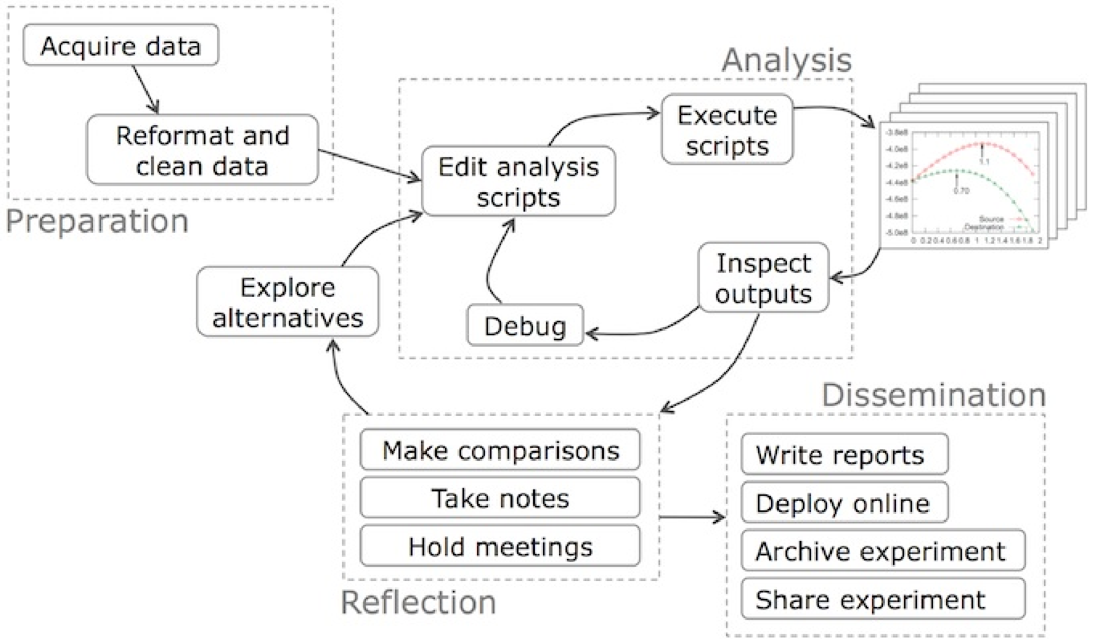
We touched on this way back in LEC01, but let's return now that we have started to dig into small bits of material.

What is involved in data science? What does a data scientist do?

Here's one of the better illustrations I've seen (courtesy ACM):

```
[2]: Image(filename='rp-overview.jpg')
```

```
[2]:
```



(Input and output with two loops: reflection outer loop; analysis inner loop.)

## A typology of data

1. Stevens, *Science*, 1946
2. `http://en.wikipedia.org/wiki/Level_of_measurement`

**Data = measurements = types of variables**

Stevens claimed that all measurements in science fall into four categories that he named:

1. Nominal,
2. Ordinal,
3. Interval,
4. Ratio.

This is somewhat different than data types (integer, float, string) in a computer program. The notion is: what do the data *mean*?

The type of data we have dictates what calculations and statistics we can apply. Let's dig in to learn what these are.

- Example statistic: **central tendency** (we'll discuss these for each level of measurement)

### Nominal data

Nominal data are qualitative. It can be used for classification but can't be ordered. Examples:

- Gender
- Race
- Genre
- Species

- Language
- Names of friends

Most mathematical operations are **not meaningful** for ordinal data. It doesn't make much sense to compute english + spanish, or science fiction * comedy, etc.

- Equality makes sense: English $\neq$ Spanish.

**Central Tendency**

- Without addition we can't generally say the *average* (precisely, arithmetic mean) is meaningful. Although we can ask what is the most common item in the data, aka the mode.

## Ordinal Data

Similar to nominal data but with a **scale** or **ordering**. Examples:

- Healthy, Sick
- Completely Agree, Mostly Agree, Mostly Disagree, Completely Disagree

An ordering lets us measure not just equality, but that, e.g., Healthy $>$ Sick.

**Central Tendency** With an ordering we can meaningful compute the "middle" of the data. This is the Median.

- The **median** is found by ordering all of the data from smallest to largest and picking the *middle* value. This is also known as the **50th percentile**.

## Interval Data

Quantitative, numerical data, often continuous. Interval data capture a meaningful notion of *difference*. These data are typically numbers on a scale with a relative/arbitrary zero point. Examples:

- Dates: 200 BC to 2015 AD.
- Temperatures: 10C, 20C
- Latitude

It's meaningful to say 2015 AD - 200 BC = 2215 years. 20C is 10 degrees warmer than 10C, but it's not **twice** as warm.

- Addition and subtraction are meaningful, as are equality and ordering (>, <). Multiplication and division are not.

**Central Tendency** With addition being meaningful for these data, we can now compute the typical (arithmetic) mean. The median and mode are also meaningful here.

## Ratio Data

Ratio data is also quantitative and numeric, but it's defined relative to some **base quantity** or unit. That's what the "ratio" means. Examples:

- Time measured in seconds or years
- Length measured in meters, inches, etc.
- Area in square meters
- Electric charge
- Mass

Ratio data is often what we think of when we mean "numerical data". Notice how it's subtly different from interval data:

- **dates** are interval data, **durations** are ratio data

Addition/Subtraction, Multiplication/Division, Equality/Inequality and Ordering are all meaningful for ratio data.

**Central Tendency** The mean, median and mode, the most common measures of tendency, are all meaningful here.

- In addition to the typical arithmetic mean, the geometric and harmonic means are sometimes meaningful here.

Some aspects of this grouping are perhaps controversial. Is there a difference between categorical and nominal data?

Take away: Any of these data can be and are represented on the computer with numbers, but it's important to understand what quantities they capture so you can compute meaningful information from those numbers. Data have units!

---

## Storing data

Suppose we have data from some source (experiment, log books, a website), how do we keep it on our computer?

Let's go over different ways to store data. Often the type of data dictates the easiest format (or the most efficient, which is not always the easiest).

### Flat files, text files

The **humble text file**. Often the simplest choice, most compatible between different code or programs, and often (disk) space efficient.

For simple tabular data, especially numeric values, a plain file of *space-separated* **columns** and *line-separated* **rows** is natural:

```
0.884936   0.605310   0.400784   0.797264
0.240286   0.422527   0.727743   0.928229
0.708107   0.259351   0.939297   0.084148
0.861638   0.075969   0.5493     0.706755
```

(Sometimes multiple spaces are used to align columns visually, but often not.)

Sometimes you have a **header row** describing the columns:

```
Lat        Lon        Height     Speed
0.884936   0.605310   0.400784   0.797264
0.240286   0.422527   0.727743   0.928229
0.708107   0.259351   0.939297   0.084148
0.861638   0.075969   0.5493     0.706755
```

So maybe this is wind speed data. (Notice how we still have no idea about the units!)

Perhaps some of our wind stations are broken. It's common to encode missing data (or missing *fields*), with `nan` (meaning, "not-a-number") or sometimes `NA`:

```
Lat        Lon        Height     Speed
0.884936   0.605310   0.400784   nan
0.240286   0.422527   0.727743   0.928229
0.708107   0.259351   0.939297   0.084148
```

```
0.861638    0.075969    0.5493      nan
```

This way you still have four columns in the file. * Often your job will be to figure out how to **insert** `nan`'s properly.

---

What happens if you have this file:

```
Lat         Lon         Height      Speed
0.884936    0.605310    0.400784    nan
0.240286    0.422527    0.727743    0.928229
0.708107    0.259351    0.939297    0.084148
0.861638    0.075969    nan         1.209123
```

but no `nan` used to mark missing entries? You could get this:

```
Lat         Lon         Height      Speed
0.884936    0.605310    0.400784
0.240286    0.422527    0.727743    0.928229
0.708107    0.259351    0.939297    0.084148
0.861638    0.075969    1.209123
```

is `Height` 1.209123 for the last row?

---

You can of course mix numeric data with text in such a file:

```
Name          Age
Hilbert, D    34
Lovelace, A   32
Knuth, D      45
Hopper, G     42
```

But here we have a **problem**! If we assume *spaces* separate columns, we have **too many spaces**:

[3]:
```
data = """Name         Age
Hilbert, D   34
Lovelace, A  32
Knuth, D     45
Hopper, G    42"""

for line in data.split("\n"):
    D = line.split()
    print(D, len(D))
print("THREE COLUMNS!")
```

```
['Name', 'Age'] 2
['Hilbert,', 'D', '34'] 3
['Lovelace,', 'A', '32'] 3
['Knuth,', 'D', '45'] 3
['Hopper,', 'G', '42'] 3
THREE COLUMNS!
```

(In reality, you wouldn't put the data *inside* the code; this is just a small example. In fact, it is important practice to separate code and data as much as possible!)

This is known as **delimiter collision**. The space delimiter collided with the space separating Last name and first initial.

- We could use a different delimiter to avoid this, for example a tab (`\t`).
- Another choice would be a comma delimiter. Then we'd have a **comma-separated value** file (csv). These are very common, but here we would also collide with a comma.

Choosing your delimiters (even your newline delimiter) can be tricky; it depends on the input data. Can you guarantee a character won't be used in any fields? If not you may need to ***escape* the delimiter**. For example, "" (a space) is a delimiter, unless it's preceded by a slash:

```
Name          Age
Hilbert,\ D    34
Lovelace,\ A   32
Knuth,\ D      45
Hopper,\ G     42
```

Of course you can write code to replace the spaces, perhaps with underscores (_), so you get:

```
Name          Age
Hilbert,_D    34
Lovelace,_A   32
Knuth,_D      45
Hopper,_G     42
```

- With CSV files in particular it's very common to **quote** fields:

```
"Date","Pupil","Grade"
"25 May","Bloggs, Fred","C"
"25 May","Doe, Jane","B"
"15 July","Bloggs, Fred","A"
```

CSVs are common enough that you have a nice Python module for reading and writing them. This is especially important because writing code to deal with **quoted fields** can be a hassle. Here's an example for reference:

[4]:
```
import csv

f = open("some.csv", 'r')

reader = csv.reader(f)
for row in reader:
    print(row)

f.close()
```

```
['Title', 'Release Date', 'Director']
['And Now For Something Completely Different', '1971', 'Ian MacNaughton']
['Monty Python And The Holy Grail', '1975', 'Terry Gilliam and Terry Jones']
["Monty Python's Life Of Brian", '1979', 'Terry Jones']
['Monty Python Live At The Hollywood Bowl', '1982', 'Terry Hughes']
["Monty Python's The Meaning Of Life", '1983', 'Terry Jones']
[]
```

Looks pretty mundane, but `csv.reader` (and `csv.writer`) has options to handle different quote characters (default "'') and delimiter characters (default ',') automatically. **So handy!**

***Don't parse CSV files yourself:*** (trigger warning)

[5]:
```
from IPython.display import Image
Image(filename='stupidity.jpg')
```

[5]:

It seems so simple, but there are a surprising number of bizarre, unexpected edge cases that can occur when processing csv files.

- quoted commas, quoted newlines
- empty fields, escaped empty fields that don't look empty
- optionally quoted fields
- fields containing more csv data
- even for numbers: 10000 vs. 10,000
- internationalization (10,000.0 vs. 10.000,0 - decimal comma!)
- ...

It's a much better idea to lean on a pre-built ("battle-tested") csv parser.

## Non-tabular data

It often occurs in practice that each piece of your data looks very different. You can store these in a table, but it might be very **sparse**:

- You might have 100 unique "fields" so you need a column for each, but any given data point (row) may only use 5 fields, for example. Almost the entire file would be `nan`. And what if each row uses a *different* set of 5 fields...?

Maybe you can dream up a way to keep the names of the fields alongside each row in your file:

Look familiar? A little like a dict. But then, why make a custom format?

Instead of writing your own code for this custom format, this is precisely the case where **JSON** (or XML) shines:

- JSON (Javascript Object Notation). It should look familiar to you because it's almost exactly how we type a dictionary into Python.

**JSON**

Suppose indented below are the contents of a file. Interpreted as JSON, this file contains a list of integers:

```
[1,2,6]
```

Looks familiar?

Here's another file:

```
{'a':8,'b':'c', 'd':9}
```

Just like a `dict`!

Here's a file encoding with JSON a list of two dictionaries (associative arrays):

(Newlines and other whitespace outside of quoted strings are ignored as JSON.)

You read this into Python with something like:

```
L = json.loads( open(json_file).read() )
```

where `json_file` is the name of this hypothetical text file.

- `json.load` - load from a file handle
- `json.loads` - load from a string

**JSONL** (See also: http://jsonlines.org/)

Here's a file with *one JSON-encoded object **per line***:

Read it like this:

```
for line in open(json_file):
    D = json.loads(line.strip())
```

- Here ***each line*** of the file is a string that can be parsed into JSON, but the entire file as a single string is not JSON (Try typing `{'a':4}{'b':5}` into IPython).
- I prefer the one-line-at-a-time JSON style, especially for very big files. That way you can unpack the objects **while reading the file**.

## Flat file?

A flat file means you are forced to read it line-by-line (or really, character-by-character). You can't easily **jump** to a specific point in the file.

- This becomes important when the data are **too big** to fit into one computer's *memory*

## Databases

Another very powerful approach to storing data is with a database. That's a very broad term, so here I'm specifically talking about a **database management system**. A **DBMS** usually consists of:

1. The data itself, stored to disk (locally on your machine or over a network to another machine),
2. A software **process** on the machine with the data that reads and writes the data for you.

Users never directly touch the data, the "server" does everything for you. This means that there is another process sitting as an intermediary, which is incredibly important for managing user access and for **concurrency**.

## SQL

Database actions are usually described using a mostly-standardized language called `SQL` (Structured Query Language, often pronounced "sequel").

- SQL provides a defined format for common actions. You `CREATE` a `table`, you `INSERT` a `row` of data into it, you `SELECT` data from it, etc.
- You define the type of variable for each `field`.

Common SQL implementations are MySQL and PostGreSQL.

## MySQL examples

- ("`mysql>`" indicates a command-line prompt when you've connected to a database.)

Create a table:

```
mysql> CREATE TABLE example_table (
         id INT PRIMARY KEY,
         data VARCHAR(100),
         cur_timestamp TIMESTAMP(8)
       );
Query OK, 0 rows affected (0.00 sec)
```

Insert a row:

```
mysql> INSERT INTO example_table (data)
            VALUES ('The time of creation is:');
Query OK, 1 row affected (0.00 sec)
```

Select all rows (only one for now):

```
mysql> SELECT * FROM example_timestamp;
+----+--------------------------+---------------------+
| id | data                     | cur_timestamp       |
+----+--------------------------+---------------------+
|  1 | The time of creation is: | 1997-08-29 02:14:00 |
+----+--------------------------+---------------------+
1 row in set (0.00 sec)
```

Select statements are **powerful**:

```
mysql> SELECT priceCat, productName FROM MyProducts
        WHERE price >= 1000 AND price < 2000;
```

The real power of a database comes from an **index**.

- Naively, to get every row where `1000 <= price < 2000` you need to look at every row in the entire table and ask if price meets your selection criteria. This **linear search** becomes very slow if you are doing it often on very large databases.
- Instead, an **index speeds** this up dramatically.
  - You can think of index like the index of a book: if you want to find all the pages where "histogram" occurs, instead of reading through the entire book, you flip to the index, jump to the "H" section, scan down to "histogram" and see a list of page numbers.
- A database index is actually built using special **tree structures** (typically a B-tree or B+ tree). Combining multiple indexes let you speed up your **queries**.

**Pros and Cons**

A database is not automatically the best choice for your problem!

There's a lot of complexity involved in learning how to work with a database, setting up a new database, network access may be slow, reading from disk may be slow, so why do it if everything fits?

**Even the most powerful indexing mechanism may not be worthwhile:**

For example, imagine you have a social network and you want get all the friends of a particular person. An index will speed this up dramatically:

select usera,userb from friends where usera == "John Doe";

```
mysql> SELECT id,usera,userb FROM net WHERE usera == "John Doe";
+----+------------------------------+
| id | usera    | userb             |
+----+------------------------------+
|  1 | John Doe | Alice Smith       |
|  2 | John Doe | Bob Johnson       |
|  3 | John Doe | Berenice Smithers |
+----+------------------------------+
3 row in set (0.01 sec)
```

The index made it super fast to jump to each "John Doe" row. **You needed to touch only a small piece of the full data**.

- But what if you want to get all the friends of **every** user. The index won't help you, you need to look at every row eventually, and if you do it badly the index can actually slow you down!

**Databases are incredibly powerful when:**

- The data is too big to fit into memory, at least on one computer,
- You need to access only a small portion of data at any given time,
- When you are constantly rewriting/replacing data values,
- **Multiple people are writing** to the database at once.

**Flat files are often better when:**

- The data can fit into memory (RAM is always faster than disk),
- You only need to add new data,
- You will touch every piece of data every time you compute something,

Every problem is different, so there are not hard-and-fast guidelines here. But I often find that, given the time and complexity to set up a database, the disadvantages often (in this class, probably **always**) outweigh the benefits.

## Other file types

Occasionally you run into other formats. For example, **binary** data is used for images and also for numerical data that needs to be incredibly space efficient.

- A plain-text file is easy to read and write, but wasteful in the sense that you are *encoding* a number "1.234" as letters "1", ".", "2", "3", "4". A binary data format can be defined to use less space.
- This can be very important for HUGE datasets, like computational fluid dynamics simulations, but often times disk space is cheap enough that it's not worth the effort.

Example numeric binary formats include MATLAB's `.mat` files, and HDF5 files (Hierarchical Data Format).

- HDF5 files are actually much more than general than just numeric data.

Geographic data (maps, landmass shapes, rivers, building outlines) are often stored in a huge variety of GIS data formats, both plain text and binary.

- Geographic data can be *very annoying* to deal with!

---

### How to choose the file/storage format that works for you

This can be incredibly challenging. The first questions to ask:

- How big is the data? Will it fit into local memory (< 1-2 GB)? or not?
- How heterogeneous is the data? Does every data point consist of the same three numbers (time vs. voltage & current) or are the fields potentially very different (twitter data)
- Can you easily describe how to represent (or think about) each piece of data? (For example: "each row is a list of friends.")
- Will you need to do a lot of **cleaning** to make sure that, e.g., all `YEAR`s are four-digit numbers? Will you frequently be replacing portions of the data over time?
- What do your colleagues want you to use? Your **boss**?

My personal advice is to choose the absolute simplest format that meets your needs. Although this can take some care and experience to judge.

## Takeaways

#### The big DS picture

- We are tackling pieces of the Data Science **pipeline**, here in class and in the readings and homeworks. The big picture emerges.
- Data have meaning. Often that meaning is associated with the **units** of the data (Stevens' levels of measurement). Use that meaning to choose appropriate (meaningful) statistics.

#### Storing data

- We are tackling pieces of the Data Science **pipeline**, here in class and in the readings and homeworks. The big picture emerges.
- CSVs look simple but that can be **deceptive** for arbitrary data. Lots of *free-form* text? Don't parse the csv fields yourself. Use a library!
- There is an art to choosing appropriate file/storage formats. Sounds simple, but can be a mess. How flexible should you be? What other constraints (prior code, bosses) limit your choices?