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

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')

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

DS1 Lecture 06

Jim Bagrow

Last time:

1. Data science "pipeline"
    - Overview, the big picture
2. Typology of data
    - Stevens' Levels of Measurement

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

Today's plan:

1. Storing data
    - Text files (CSV), JSON, XML, databases
    - Delimiter collisions
    - How to choose the right format?
2. Retrieving data
    - A case study, with some important asides [time permitting]

Of course, these two are connected: once you've retrieved some data, you will want to store it!

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**:

[2]:
```
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:

[3]:
```
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)

[4]:
```
from IPython.display import Image
Image(filename='stupidity.jpg')
```

[4]:

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 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 a JSON list of two dictionaries (associative arrays) (newlines and whitespace outsie quoted strings are ignored):

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 (Try typing "{'a':4}{'b':5}" (without quotes) 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.

Retrieving data

The pipeline

Recall from last time:

Now we will cover the first phase:

But we will also get into a bit of this piece:

We should never forget our final goal: communicating our insights to others!

Accessing Web APIs

Case study: open currency exchange

As an data-collection example, suppose we want to find out how currencies compare to one another over time. In other words, let's plot a time series of exchange rates.

There's a nice, free website called https://openexchangerates.org. They provide a nice API to get exchange rate data. Let's use this.

- You need to register with them to get an **APP ID**. This lets them track how often you call their website and block you if you do too much (this is known as rate limiting).

The APP ID is a string of 32 characters. I've got mine saved by itself in a text file which the python will load:

[5]:
```
app_id = open("api_id.txt").read().strip()
```

Now let's download something and see what we get. Their docs help us see how to build a URL.

[6]:
```
# from their docs:
# http://openexchangerates.org/api/latest.json?app_id=YOUR_APP_ID

# build a url from pieces:
base_url = "http://openexchangerates.org/api"
id_str   = "app_id={}".format(app_id)
URL = "{}/historical/2011-10-18.json?{}".format(base_url, id_str)

print(URL[:-30], "...")
```

```
http://openexchangerates.org/api/historical/2011-10-18.json?app_id=21 ...
```

[7]:
```
import urllib.request

# `connection` behaves like a file even though it's a webpage
connection = urllib.request.urlopen(URL)
text = connection.read().decode("utf-8") # behaves like a file
connection.close()

# now print the beginning and ending of the text:
print(text[:1000])
print("[...]")
print(text[-300:])
```


```
{
  "disclaimer": "Usage subject to terms: https://openexchangerates.org/terms",
  "license": "https://openexchangerates.org/license",
  "timestamp": 1318953600,
  "base": "USD",
  "rates": {
    "AED": 3.67285,
    "AFN": 48.325965,
    "ALL": 102.607855,
    "AMD": 376.327731,
    "ANG": 1.77665,
    "AOA": 94.851761,
    "ARS": 4.215038,
    "AUD": 0.979142,
    "AWG": 1.79025,
    "AZN": 0.786155,
    "BAM": 1.429934,
```

```
    "BBD": 2,
    "BDT": 75.987773,
    "BGN": 1.430108,
    "BHD": 0.37653,
    "BIF": 1231.30548,
    "BMD": 1,
    "BND": 1.272581,
    "BOB": 7.013496,
    "BRL": 1.767354,
    "BSD": 1,
    "BTN": 49.334603,
    "BWP": 7.340381,
    "BYR": 4395.431805,
    "BZD": 1.99315,
    "CAD": 1.018634,
    "CDF": 915.22783,
    "CHF": 0.900405,
    "CLF": 0.021176,
    "CLP": 510.174179,
    "CNY": 6.3813,
    "COP": 1894.791035,
    "CRC": 510.707928,
    "CVE": 80.624452,
    "CZK": 18.206884,
    "DJF": 177.721,
    "DKK": 5.434641,
    "DOP": 38.360152,
    "DZD": 73.7
[...]
618,
    "VEF": 4.29465,
    "VND": 20919.570165,
    "VUV": 91.508054,
    "WST": 2.310598,
    "XAF": 480.262994,
    "XCD": 2.693174,
    "XDR": 0.635175,
    "XOF": 479.520484,
    "XPF": 87.253144,
    "YER": 214.263011,
    "ZAR": 8.044653,
    "ZMK": 5060.311287,
    "ZWD": 15180.722235
  }
}
```


OK, this is nice we've got some data in a human-readable format.

- More JSON (Javascript Object Notation)!
- This data format has become **very popular recently** because it's not only how you write a Python dict but also a **Javascript object**. This means a web browser can trivially parse a JSON string. Other formats, like XML, require real parser code.

JSON can actually represent more than dicts, like a list of dicts (it does not support *sets* though):

Great!

This means we can take the text from that website and run it through `json.loads` and we have a nice accessible python dict:

[8]:
```
data = json.loads(text)
print(type(data))
print(data.keys())
```

```
<class 'dict'>
dict_keys(['disclaimer', 'license', 'timestamp', 'base', 'rates'])
```

Sweet. Now we see there's a timestamp key. What's that give us?

[9]:
```
print(data["timestamp"])
```

```
1318953600
```

What the!!!!!

- Any idea what that could be?

Dealing with dates and times

Timestamps can be surprisingly tricky to deal with in programs. For example:

- How many days are between April 21, 1986 and today?

To answer this properly, you need full information on the calendar, including leap years. And don't get started on **time zones**.

datetime

Fortunately, Python has builtin support for date and time data. It's not trivial to use but it works.

Let's answer the above question. `datetime` gives us useful date objects, `date`, `time`, and `datetime`. These let us store a date, a time of day, or both, respectively.

[10]:
```
import datetime

D1 = datetime.date(1986, 4, 21)
T1 = datetime.time(12,0,0) # noon
DT = datetime.datetime(1986, 4, 21, 12, 15, 0)

# Typically you want to work with datetime because you can
# omit the time values and then it defaults to midnight.
D = datetime.datetime(1986,4,21)
```

Once you have a `datetime` object you can do fancy things with it:

[11]:
```
print("The year was %i and the day was %i." % (D.year, D.day))

print("The day of the week was %i." % (D.weekday()))
print("(Monday = 0, ..., Sunday = 6.)")
```

```
The year was 1986 and the day was 21.
The day of the week was 0.
(Monday = 0, ..., Sunday = 6.)
```

More importantly, these objects support **math operations that are meaningful for time**:

[12]:
```
Dnow = datetime.datetime.now()

print(Dnow - D)
```

```
12196 days, 20:10:15.780591
```

What this actually did is create another data object, called a `timedelta` object:

[13]:
```
dt = Dnow - D
print(type(dt))
print("There are %i days between then and now." % dt.days)
```

```
<class 'datetime.timedelta'>
There are 12196 days between then and now.
```

A `timedelta` will let us incorporate time intervals:

[14]:
```
interval = datetime.timedelta(days=100,hours=12) # 100.5 days

soon = datetime.datetime.now() + interval # addition!

interval_days = interval.total_seconds()/3600.0/24

print("In %0.1f days it will be %s." % (interval_days, soon))
```

```
In 100.5 days it will be 2019-12-21 08:10:15.894830.
```

OK, the real janitorial work comes when reading and writing **timestamps**.

- We need to be able to understand how the string `"2012-04-26"` contains the same data as the string `"April 26, 2012"`.

`datetime` provides us tools to read and write such timestamps. Let's first define two different timestamps and a timedelta

[15]:
```
ts1 = "2012-04-26"
ts2 = "January 5, 1978"
```

We are now going to use a function to parse a string for a time given that string and another string representing a **time format**.

- This function is called `strptime` (To help remember it, I read it as "**str**ing __p__arse **time**").

Here we go.

[37]:
```
d1 = datetime.datetime.strptime( ts1, "%Y-%m-%d" )
print(d1)
print(d1 + datetime.timedelta(days=-7))
```

```
2012-04-26 00:00:00
2012-04-19 00:00:00
```

The string `"%Y-%m-%d"` encodes the timestamp format we were looking for. A four-digit year (`%Y`), a dash (`-`), a two-digit month number (`%m`), another dash, and then a day number (`%d`).

Now `ts2` incorporates the name of a month, so that format string is a little different (`%B` means the full month name).

[17]:
```
d2 = datetime.datetime.strptime( ts2, "%B %d, %Y" )
print(d2)
print(d2 - datetime.timedelta(days=-7))
```

```
1978-01-05 00:00:00
1978-01-12 00:00:00
```

There's a **huge number of ways to build a format string**. I always have to look them up: * https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

Parallel to `strptime` is another function, `strftime` ("**str**ing __f__ormat **time**") that does the opposite actions: take a `date` or `datetime` and print it out according to a timestamp format string

[18]:
```
s_before = "Jan 19, '89"
d = datetime.datetime.strptime("Jan 19, '89", "%b %d, '%y")
s_after  = d.strftime("%Y-%m-%d")
print(s_before, "--->", s_after)
```

```
Jan 19, '89 ---> 1989-01-19
```

This is particular conversion is useful because different data sources encode times in different ways. Some formats are easy for humans to read, but I like the `%Y-%m-%d %H:%M:%S` format timestamp because it *sorts nicely*.

- This is also the international standard for transmitting dates and times! (ISO 8601)
- When you are creating timestamps, **always use the ISO standard format**.

OK, now, what about our original timestamp we retrieved?

[19]:
```
print(data["timestamp"])
```

```
1318953600
```

What format is that?

Aside within an aside: the epoch!

Sometimes you see a date that looks weird:

[20]:
```
import time # another time module!

print(time.time()) # what the heck!
```

```
1568247016.291855
```

This function is another way to get the *current time* but it's encoded in a **numeric** format:

- **the number of seconds since the epoch**.

Let's explore:

[21]:
```
t = time.time()
y = t / 60 / 60 / 24 / 365 # oops, leap years!
print(y)
```

```
49.728786667630054
```

OK, what the heck happened back then?

[22]:
```
days = time.time() / 60 / 60 / 24
print(datetime.datetime.now() - datetime.timedelta(days=days))
```

```
1969-12-31 20:00:00.000041
```

The epoch ("epoch" means "reference date") is the UNIX epoch, Jan 1, 1970.

- `time.time()` returns the number of seconds since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds.

Why 1970?

Because of these guys:

These numeric timestamp formats were very useful when it was too expensive to have a complex library like `datetime`.

This timestamp format is popular enough that `datetime` objects have a builtin converter method:

[23]:
```
t = datetime.datetime.fromtimestamp( data["timestamp"] )

print(t)
```

```
2011-10-18 12:00:00
```

Whew!!!!

Next time

We will continue this currency exchange rate example next time.

Takeaways

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?

Retrieving data

- Acquiring data means understanding the input format as best we can (in this case, by studying the web service's API docs) and by using the appropriate tools within our own programs to handle that data (in this case, just some basic lists and dicts).
  - Simple **sanity checks** are a key piece to understanding a new dataset!
- As an aside, we started learning a lot (but not everything) about working with dates and times
  - … timezones!