[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 07

Jim Bagrow

Last time:

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
   - Working with dates and times (datetimes!)
     - `from datetime import datetime`
     - `strptime` and `strftime` https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

Today's plan:

1. Continue with data retrieval case study
2. More on tabular data [slides]

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

Retrieving data

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:

[2]:
```
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.

[3]:
```python
# 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=77 ...
```

[4]:
```python
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:

[5]:
```python
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?

[6]:
```
print(data["timestamp"])
```

```
1318953600
```

What the!!!!!

- Any idea what that could be?
- **From last time:** The number of seconds since January 1, 1970 (the UNIX *epoch*)

Back to the data:

There's also `base` and `rates` keys. Those are the actual exchange rate data:

[7]:
```
print(data["base"])
print(type(data["rates"]))
print( list(data["rates"].keys())[:5]) # print first 5 keys only
```

```
USD
<class 'dict'>
['AED', 'AFN', 'ALL', 'AMD', 'ANG']
```

`base` tells us what currency the exchange rate is relative to. Meanwhile, `rates` is another dict, keyed by three-letter currency name.

Sanity check:

[8]:
```
print(data["rates"]["USD"])
```

```
1
```

Makes sense, the conversion for USD should always be 1 since the base was USD.

- This simple calculation (the ***sanity check***), is one of the single most important things you can do when analyzing data!
- **I'm expecting to see lots of santiy checks in your work**

Now, having a `data` dict like this is a little verbose compared to a table or CSV file.

- A CSV file for exchange rates makes a lot of sense but many data do not fit into a nice regular form like that.
- Sending JSON "over the wire" and using dictionary keys makes it easy for us to keep track of what number correspond to what unit of measurement.

Putting it all together

We want to get the currency exchange rates over a long time period. Let's go with *monthly rates*. * We'll cheat a little though. For this example, we'll just retrieve the exchange rate on the first day of each month.

Let's make a for loop over all years and months between 1999 and 2013, download the JSON exchange data for each month, and save it to a big dict keyed by (year,month) tuples:

- (Do you think this is the best design for retrieving and storing these data?)

[9]:
```
time2data = {}
for year in range(1999,2013+1):
    for month in range(1,12+1):

        # format the timestamp
        f = "{}-{:02d}-01.json".format(year,month)

        # GET variable for app id
        ids = "app_id={}".format(app_id)

        # assemble URL for this year,month:
        url = "{}/historical/{}?{}".format(base_url,f,ids)

        # debug:
        #print(f, ids, url)
        #continue

        # retrieve and process:
        json_str = urllib.request.urlopen(url).read().decode('utf-8')
        data = json.loads(json_str)

        # store in our new dict of dicts:
        time2data[(year,month)] = data
```

[10]:
```
len(time2data)
```

[10]:
```
180
```

Not bad, and it only took a few seconds to run. Now we can look at some data:

[11]:
```
list_ex = []
list_ts = []

for y_m in sorted(time2data.keys()): # sorted() makes sure we loop
                                     # over the dict in time order
    list_ts.append(y_m)

    data = time2data[y_m]
    exch_eur = data["rates"]["EUR"]
    list_ex.append(exch_eur)
```

This gives us a list of (year,month) tuples and a list of EUR->USD exchange rates (since `base` is always USD).

[12]:
```
print(list_ts[:4])
print(list_ex[:4])
```

```
[(1999, 1), (1999, 2), (1999, 3), (1999, 4)]
[0.853515, 0.884111, 0.917058, 0.927253]
```

Let's make a plot!!!

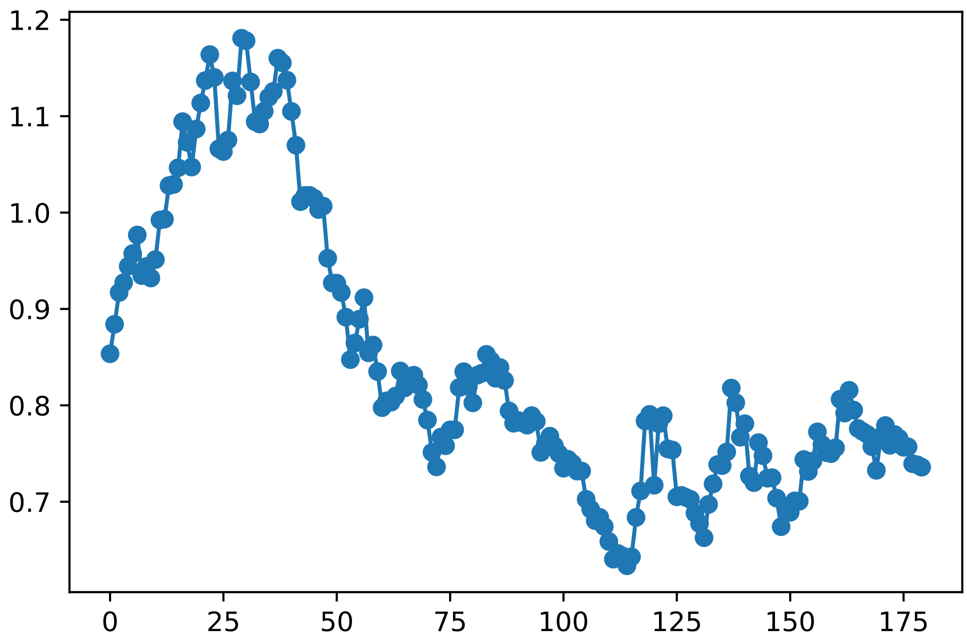
We've got a time series, a natural x-variable would be the **time** of the exchange rate and the y-variable would be the **value** of the exchange rate.

- Hmm, How to plot the times...

1. Quick-and-dirty approach: replace them with a number (index)!

[13]:
```
import matplotlib.pyplot as plt

plt.plot(list_ex, 'o-')
plt.show()
print("\n"*10)
```



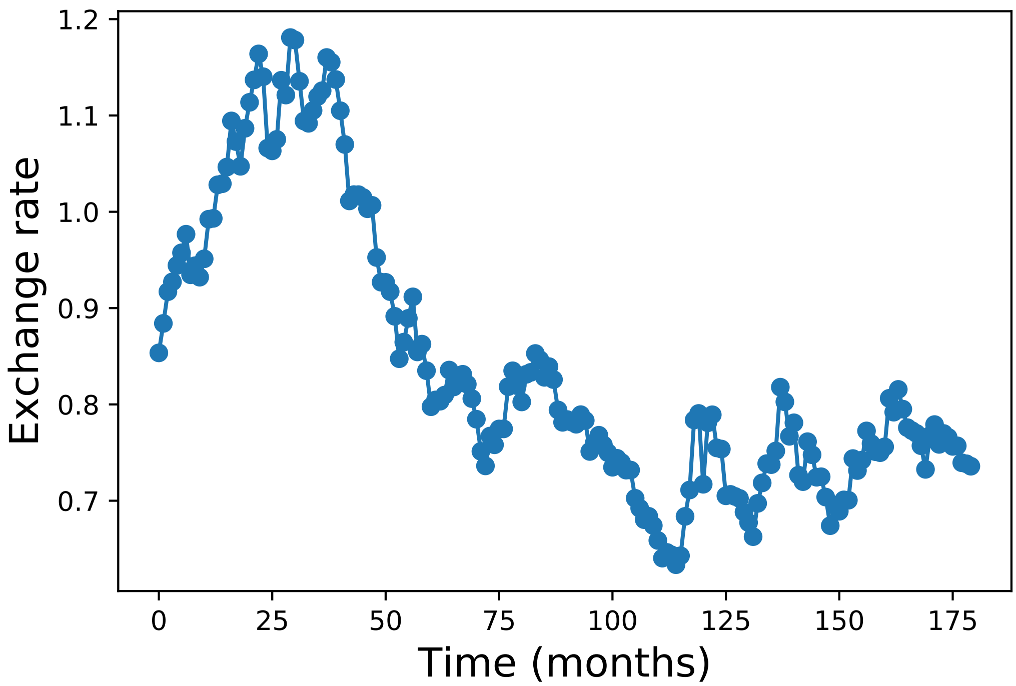
This is *poor* plot. Why?

- No labels on the axes!

[14]:
```
plt.plot(list_ex, 'o-')
```

```
plt.xlabel("Time (months)", fontsize=15)
plt.ylabel("Exchange rate", fontsize=15)


plt.show()
print("\n"*10)
```



This plot is telling us a story. We got the whole thing at this point. But we are not always the final audience.

- We may be communicating this story to someone else. Someone who did not write the code above.

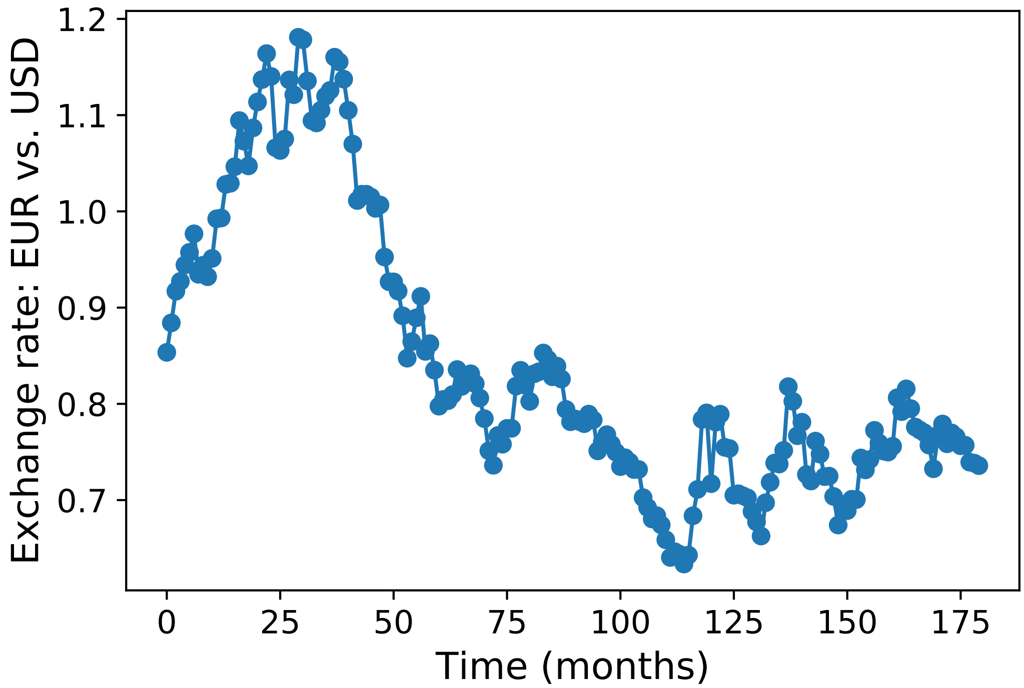
And in **this** context: This is *still* a poor plot. Why?

- Labels aren't very useful!

[15]:
```
plt.plot(list_ex, "o-")

# clean it up:
```

```
plt.xlabel("Time (months)", fontsize=14)
plt.ylabel("Exchange rate: EUR vs. USD", fontsize=14)
plt.tick_params(labelsize=12) # how? some googling
plt.show()
```



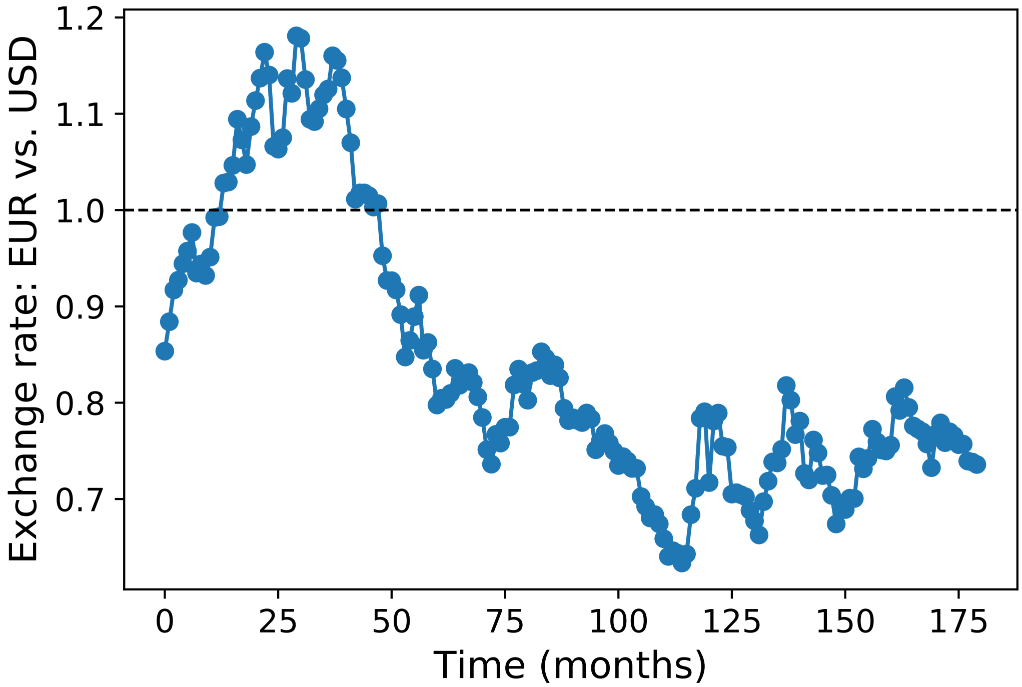
Let's grind on this a bit more. (Of course you would do this by iterating on the **same** code; but I'm using different cells for these notes.)

- Can we put in an **affordance** for interpretation? Something to guide the reader/viewer?

[16]:
```
plt.plot(list_ex, "o-")

# add horizontal bar denoting EUR = USD:
plt.axhline(1.0, linewidth=1, color='k', linestyle='dashed')

# clean it up:
plt.xlabel("Time (months)", fontsize=14)
plt.ylabel("Exchange rate: EUR vs. USD", fontsize=14)
plt.tick_params(labelsize=12) # how? some googling
plt.show()
```



Something simple to help communicate what the exchange rate *means*.

- After doing something like this, you need to ask yourself: was it necessary? Maybe, maybe not!

Label the time axis properly

If we have an audience beyond ourselves (including future us!), we want to **sweat the details** to make their job as easy possible.

- That x-axis is not the easiest thing to read. Let's fix it!

The plotting library `matplotlib` gives us a very nice tool for making time series plots. It works using Python's so-called `datetime` object.

- We need to convert our ad hoc time unit, the `(year, month)` tuples, into something more proper:

[17]:
```
import datetime

list_dt = []
for yr,mo in list_ts:
    dt = datetime.datetime(year=yr,month=mo,day=1) # !!
    list_dt.append(dt)
```

Now we can use matplotlib's tools.

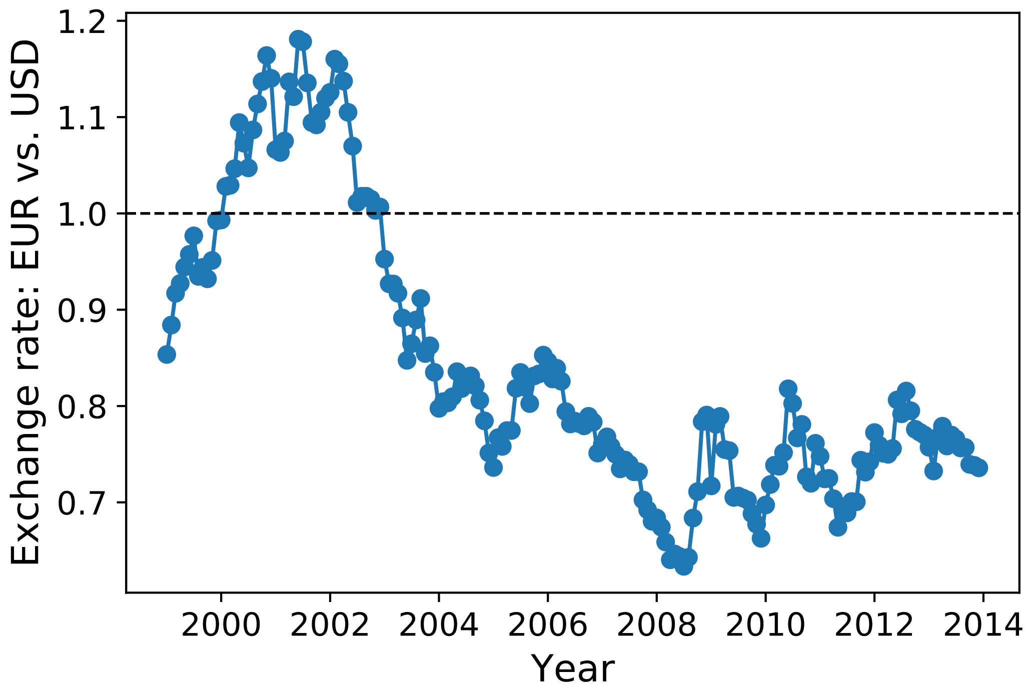
[18]:
```
import matplotlib.dates
dates = matplotlib.dates.date2num(list_dt)
```

We can now plot exactly as before except we use the `dates` list for an x-variable, and we use `plt.plot_date` instead of `plt.plot`:

[19]:
```
plt.plot_date(dates, list_ex, "o-") # note plot_date !


# add horizontal bar denoting when EUR = USD:
plt.axhline(1.0, linewidth=1, color='k', linestyle='dashed')

# clean it up:
plt.xlabel("Year", fontsize=14)
plt.ylabel("Exchange rate: EUR vs. USD", fontsize=14)
plt.tick_params(labelsize=12) # how? some googling
plt.show()
```



Other sources of data

Many other protocols exist besides web APIs for requesting data. As one example:

Email

IMAP (*Internet Message Access Protocol*) - Standard for storing, sending, receiving and other accessing email messages. IMAP is very common and well established so (of course!) there's a Python library for it:

```
import imaplib
import time

IMAP_SERVER = 'imap.gmail.com'
USERNAME = 'username@gmail.com'
PASSWORD = 'password' # not very secure...

def download_emails(ids):
```

```
    client = imaplib.IMAP4_SSL(IMAP_SERVER)
    client.login(USERNAME, PASSWORD)
    client.select()
    for i in ids:
        print(f'Downloading mail id: {i.decode()}')
        _, data = client.fetch(i, '(RFC822)')
        with open(f'emails/{i.decode()}.eml', 'wb') as f:
            f.write(data[0][1])
    client.close()
    print(f'Downloaded {len(ids)} mails!')

start = time.time()

client = imaplib.IMAP4_SSL(IMAP_SERVER)
client.login(USERNAME, PASSWORD)
client.select()
_, ids = client.search(None, 'ALL')
ids = ids[0].split()
ids = ids[:100]
client.close()

download_emails(ids)
print('Time:', time.time() - start, "seconds")
```

```
Time: 35.65300488471985 seconds
```

(Courtesy: Floyd Hub)

(Aside: parallel processing)

This email example is also pretty cool, because it can be **parallelized**:

```
import imaplib
[...]
client.close()

from concurrent.futures import ThreadPoolExecutor # !!!

number_of_chunks = 10
chunk_size = 10
executor = ThreadPoolExecutor(max_workers=number_of_chunks)
futures = []
for i in range(number_of_chunks):
    chunk = ids[i*chunk_size:(i+1)*chunk_size]
    futures.append(executor.submit(download_emails, chunk))

for future in concurrent.futures.as_completed(futures):
    pass
print('Time:', time.time() - start, "seconds")
```

```
Time: 9.841094255447388 seconds
```

- One example of running multiple "copies" of the code at the same time, in this case multiple downloaders.

Takeaways

- 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!
- We learned a lot (but not everything) about working with dates and times
  - … timezones!
- Simple plotting, but always with an eye to communication!