Homework 1

UIC CS 418, Spring 2022

*According to the **Academic Integrity Policy** of this course, all work submitted for grading must be done individually, unless otherwise specified. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml (https://dos.uic.edu/conductforstudents.shtml).*

Due Date

This assignment is due at 11:59pm on February 4, 2022. All parts of the assignments are due at the same time. If any segment of the assignment is submitted late, the late submission policy applies for the whole assignment. Instructions on how to submit it to Gradescope are given at the end of the notebook and should be followed carefully.

Part 1 (50% of HW1): Data processing with pandas

In this homework you will see examples of some commonly used data wrangling tools in Python. In particular, we aim to give you some familiarity with:

- Slicing data frames
- Filtering data
- Grouped counts
- Joining two tables
- NA/Null values

Part 1: Practice (20%)

This part of the homework is graded manually based on showing the correct outputs after executing each step.

Setup

You need to execute each step (run each Cell), in order for the next ones to work. First, import necessary libraries:

In [5]:

```
import pandas as pd
import numpy as np
```

The code below produces the data frames used in the examples:

In [6]:

```
heroes = pd.DataFrame(
    data={'color': ['red', 'green', 'black',
                    'blue', 'black', 'red'],
          'first_seen_on': ['a', 'a', 'f', 'a', 'a', 'f'],
          'first_season': [2, 1, 2, 3, 3, 1]},
    index=['flash', 'arrow', 'vibe',
           'atom', 'canary', 'firestorm']
)

identities = pd.DataFrame(
    data={'ego': ['barry allen', 'oliver queen', 'cisco ramon',
                  'ray palmer', 'sara lance',
                  'martin stein', 'ronnie raymond'],
          'alter-ego': ['flash', 'arrow', 'vibe', 'atom',
                        'canary', 'firestorm', 'firestorm']}
)

teams = pd.DataFrame(
    data={'team': ['flash', 'arrow', 'flash', 'legends',
                   'flash', 'legends', 'arrow'],
          'hero': ['flash', 'arrow', 'vibe', 'atom',
                   'killer frost', 'firestorm', 'speedy']})
```

Pandas and Wrangling

For the examples that follow, we will be using a toy data set containing information about superheroes in the Arrowverse. In the `first_seen_on` column, `a` stands for Archer and `f` , Flash.

In [7]:

```
heroes
```

Out[7]:

	color	first_seen_on	first_season
flash	red	a	2
arrow	green	a	1
vibe	black	f	2
atom	blue	a	3
canary	black	a	3
firestorm	red	f	1

In [8]:

```
identities
```

Out[8]:

	ego	alter-ego
0	barry allen	flash
1	oliver queen	arrow
2	cisco ramon	vibe
3	ray palmer	atom
4	sara lance	canary
5	martin stein	firestorm
6	ronnie raymond	firestorm

In [9]:

```
teams
```

Out[9]:

	team	hero
0	flash	flash
1	arrow	arrow
2	flash	vibe
3	legends	atom
4	flash	killer frost
5	legends	firestorm
6	arrow	speedy

Slice and Dice

Column selection by label

To select a column of a `DataFrame` by column label, the safest and fastest way is to use the `.loc` method. General usage looks like `frame.loc[rowname,colname]` . (Reminder that the colon `:` means "everything"). For example, if we want the `color` column of the `heroes` data frame, we would use :

In [10]:

```
heroes.loc[:, 'color']
```

Out[10]:

```
flash          red
arrow        green
vibe         black
atom          blue
canary       black
firestorm      red
Name: color, dtype: object
```

Selecting multiple columns is easy. You just need to supply a list of column names. Here we select the color and value columns:

In [11]:
```
heroes.loc[:, ['color', 'first_season']]
```

Out[11]:

	color	first_season
flash	red	2
arrow	green	1
vibe	black	2
atom	blue	3
canary	black	3
firestorm	red	1

While .loc is invaluable when writing production code, it may be a little too verbose for interactive use. One recommended alternative is the [] method, which takes on the form frame['colname'].

In [12]:
```
heroes['first_seen_on']
```

Out[12]:
```
flash        a
arrow        a
vibe         f
atom         a
canary       a
firestorm    f
Name: first_seen_on, dtype: object
```

Row Selection by Label

Similarly, if we want to select a row by its label, we can use the same .loc method.

In [13]:
```
heroes.loc[['flash', 'vibe'], :]
```

Out[13]:

	color	first_seen_on	first_season
flash	red	a	2
vibe	black	f	2

If we want all the columns returned, we can, for brevity, drop the colon without issue.

In [14]:
```
heroes.loc[['flash', 'vibe']]
```

Out[14]:

	color	first_seen_on	first_season
flash	red	a	2
vibe	black	f	2

General Selection by Label

More generally you can slice across both rows and columns at the same time. For example:

In [15]:

```
heroes.loc['flash':'atom', :'first_seen_on']
```

Out[15]:

	color	first_seen_on
flash	red	a
arrow	green	a
vibe	black	f
atom	blue	a

Selection by Integer Index

If you want to select rows and columns by position, the Data Frame has an analogous `.iloc` method for integer indexing. Remember that Python indexing starts at 0.

In [16]:

```
heroes.iloc[:4,:2]
```

Out[16]:

	color	first_seen_on
flash	red	a
arrow	green	a
vibe	black	f
atom	blue	a

Filtering with boolean arrays

Filtering is the process of removing unwanted material. In your quest for cleaner data, you will undoubtedly filter your data at some point: whether it be for clearing up cases with missing values, culling out fishy outliers, or analyzing subgroups of your data set. For example, we may be interested in characters that debuted in season 3 of Archer. Note that compound expressions have to be grouped with parentheses.

In [17]:

```
heroes[(heroes['first_season']==3) & (heroes['first_seen_on']=='a')]
```

Out[17]:

	color	first_seen_on	first_season
atom	blue	a	3
canary	black	a	3

Problem Solving Strategy

We want to highlight the strategy for filtering to answer the question above:

- **Identify the variables of interest**

    - Interested in the debut: `first_season` and `first_seen_on`
- **Translate the question into statements one with True/False answers**
    - Did the hero debut on Archer? → The hero has `first_seen_on` equal to `a`
    - Did the hero debut in season 3? → The hero has `first_season` equal to `3`
- **Translate the statements into boolean statements**
    - The hero has `first_seen_on` equal to `a` → `hero['first_seen_on']=='a'`
    - The hero has `first_season` equal to `3` → `heroes['first_season']==3`
- **Use the boolean array to filter the data**

Note that compound expressions have to be grouped with parentheses.

For your reference, some commonly used comparison operators are given below.

Symbol	Usage	Meaning
==	a == b	Does a equal b?
<=	a <= b	Is a less than or equal to b?
>=	a >= b	Is a greater than or equal to b?
<	a < b	Is a less than b?
>	a > b	Is a greater than b?
~	~p	Returns negation of p
|	p | q	p OR q
&	p & q	p AND q
^	p ^ q	p XOR q (exclusive or)

An often-used operation missing from the above table is a test-of-membership. The `Series.isin(values)` method returns a boolean array denoting whether each element of `Series` is in `values`. We can then use the array to subset our data frame. For example, if we wanted to see which rows of `heroes` had values in $\{1, 3\}$, we would use:

```
In [18]: heroes[heroes['first_season'].isin([1,3])]
```

Out[18]:

	color	first_seen_on	first_season
arrow	green	a	1
atom	blue	a	3
canary	black	a	3
firestorm	red	f	1

Notice that in both examples above, the expression in the brackets evaluates to a boolean series. The general strategy for filtering data frames, then, is to write an expression of the form `frame[logical statement]`.

Counting Rows

To count the number of instances of a value in a `Series` , we can use the `value_counts` method. Below we count the number of instances of each color.

In [19]:
```
heroes['color'].value_counts()
```

Out[19]:
```
red      2
black    2
green    1
blue     1
Name: color, dtype: int64
```

A more sophisticated analysis might involve counting the number of instances a tuple appears. Here we count $(color, value)$ tuples.

In [20]:
```
heroes.groupby(['color', 'first_season']).size()
```

Out[20]:
```
color  first_season
black  2               1
       3               1
blue   3               1
green  1               1
red    1               1
       2               1
dtype: int64
```

This returns a series that has been multi-indexed. We'll eschew this topic for now. To get a data frame back, we'll use the `reset_index` method, which also allows us to simulataneously name the new column.

In [21]:
```
heroes.groupby(['color', 'first_season']).size().reset_index(name='count')
```

Out[21]:

	color	first_season	count
0	black	2	1
1	black	3	1
2	blue	3	1
3	green	1	1
4	red	1	1
5	red	2	1

Joining Tables on One Column

Suppose we have another table that classifies superheroes into their respective teams. Note that `canary` is not in this data set and that `killer frost` and `speedy` are additions that aren't in the original `heroes` set.

For simplicity of the example, we'll convert the index of the `heroes` data frame into an explicit column called `hero` . A careful examination of the documentation (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html) will reveal that joining on a mixture of the index and columns is possible.

In [22]:
```
heroes['hero'] = heroes.index
heroes
```

Out[22]:

	color	first_seen_on	first_season	hero
flash	red	a	2	flash
arrow	green	a	1	arrow
vibe	black	f	2	vibe
atom	blue	a	3	atom
canary	black	a	3	canary
firestorm	red	f	1	firestorm

Inner Join

The inner join below returns rows representing the heroes that appear in both data frames.

In [23]:
```
pd.merge(heroes, teams, how='inner', on='hero')
```

Out[23]:

	color	first_seen_on	first_season	hero	team
0	red	a	2	flash	flash
1	green	a	1	arrow	arrow
2	black	f	2	vibe	flash
3	blue	a	3	atom	legends
4	red	f	1	firestorm	legends

Left and right join

The left join returns rows representing heroes in the `heroes` ("left") data frame, augmented by information found in the `teams` data frame. Its counterpart, the right join, would return heroes in the `teams` data frame. Note that the `team` for hero `canary` is an `NaN` value, representing missing data.

```
In [24]: pd.merge(heroes, teams, how='left', on='hero')
```

Out[24]:

	color	first_seen_on	first_season	hero	team
0	red	a	2	flash	flash
1	green	a	1	arrow	arrow
2	black	f	2	vibe	flash
3	blue	a	3	atom	legends
4	black	a	3	canary	NaN
5	red	f	1	firestorm	legends

Outer join

An outer join on `hero` will return all heroes found in both the left and right data frames. Any missing values are filled in with `NaN` .

```
In [25]: pd.merge(heroes, teams, how='outer', on='hero')
```

Out[25]:

	color	first_seen_on	first_season	hero	team
0	red	a	2.0	flash	flash
1	green	a	1.0	arrow	arrow
2	black	f	2.0	vibe	flash
3	blue	a	3.0	atom	legends
4	black	a	3.0	canary	NaN
5	red	f	1.0	firestorm	legends
6	NaN	NaN	NaN	killer frost	flash
7	NaN	NaN	NaN	speedy	arrow

More than one match?

If the values in the columns to be matched don't uniquely identify a row, then a cartesian product is formed in the merge. For example, notice that `firestorm` has two different egos, so information from `heroes` had to be duplicated in the merge, once for each ego.

In [26]:

```
pd.merge(heroes, identities, how='inner',
         left_on='hero', right_on='alter-ego')
```

Out[26]:

	color	first_seen_on	first_season	hero	ego	alter-ego
0	red	a	2	flash	barry allen	flash
1	green	a	1	arrow	oliver queen	arrow
2	black	f	2	vibe	cisco ramon	vibe
3	blue	a	3	atom	ray palmer	atom
4	black	a	3	canary	sara lance	canary
5	red	f	1	firestorm	martin stein	firestorm
6	red	f	1	firestorm	ronnie raymond	firestorm

Missing Values

There are a multitude of reasons why a data set might have missing values. The current implementation of Pandas uses the numpy NaN to represent these null values (older implementations even used `-inf` and `inf`). Future versions of Pandas might implement a true `null` value---keep your eyes peeled for this in updates! More information can be found http://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html (http://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html)

Because of the specialness of missing values, they merit their own set of tools. Here, we will focus on detection. For replacement, see the docs.

In [27]:

```
x = np.nan
y = pd.merge(heroes, teams, how='outer', on='hero')['first_season']
y
```

Out[27]:

```
0    2.0
1    1.0
2    2.0
3    3.0
4    3.0
5    1.0
6    NaN
7    NaN
Name: first_season, dtype: float64
```

To check if a value is null, we use the `isnull()` method for series and data frames. Alternatively, there is a `pd.isnull()` function as well.

```
In [28]: x.isnull() # won't work since x is neither a series nor a data frame
```

```
---------------------------------------------------------------------------
--
AttributeError                            Traceback (most recent call las
t)
/var/folders/gc/8nsbgz6j6v3112s8c9l3yrym0000gn/T/ipykernel_96013/42828388
27.py in <module>
----> 1 x.isnull() # won't work since x is neither a series nor a data fr
ame

AttributeError: 'float' object has no attribute 'isnull'
```

```
In [29]: pd.isnull(x)
```

```
Out[29]: True
```

```
In [30]: y.isnull()
```

```
Out[30]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         7     True
         Name: first_season, dtype: bool
```

```
In [31]: pd.isnull(y)
```

```
Out[31]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         7     True
         Name: first_season, dtype: bool
```

Since filtering out missing data is such a common operation, Pandas also has conveniently included the analogous `notnull()` methods and function for improved human readability.

In [32]:
```
y.notnull()
```

Out[32]:
```
0     True
1     True
2     True
3     True
4     True
5     True
6    False
7    False
Name: first_season, dtype: bool
```

In [33]:
```
y[y.notnull()]
```

Out[33]:
```
0    2.0
1    1.0
2    2.0
3    3.0
4    3.0
5    1.0
Name: first_season, dtype: float64
```

Part 1: Questions (30%)

The practice problems below use the department of transportation's "On-Time" flight data for all flights originating from SFO or OAK in January 2016. Information about the airports and airlines are contained in the comma-delimited files `airports.dat` and `airlines.dat` , respectively. Both were sourced from http://openflights.org/data.html (http://openflights.org/data.html).

Disclaimer: There is a more direct way of dealing with time data that is not presented in these problems. This activity is merely an academic exercise.

In [34]:
```
flights = pd.read_csv("flights.dat", dtype={'sched_dep_time': 'f8', 'sched_
# show the first few rows, by default 5
flights.head()
```

Out[34]:

	year	month	day	date	carrier	tailnum	flight	origin	destination	sched_dep_time	actual_de
0	2016	1	1	2016-01-01	AA	N3FLAA	208	SFO	MIA	630.0	
1	2016	1	2	2016-01-02	AA	N3APAA	208	SFO	MIA	600.0	
2	2016	1	3	2016-01-03	AA	N3DNAA	208	SFO	MIA	630.0	
3	2016	1	4	2016-01-04	AA	N3FGAA	208	SFO	MIA	630.0	
4	2016	1	5	2016-01-05	AA	N3KUAA	208	SFO	MIA	640.0	

In [220]:

```
airports_cols = [
    'openflights_id',
    'name',
    'city',
    'country',
    'iata',
    'icao',
    'latitude',
    'longitude',
    'altitude',
    'tz',
    'dst',
    'tz_olson',
    'type',
    'airport_dsource'
]

airports = pd.read_csv("airports.dat", names=airports_cols)
airports.head(3)
```

Out[220]:

	openflights_id	name	city	country	iata	icao	latitude	longitude	altitude	tz	ds
0	1	Goroka	Goroka	Papua New Guinea	GKA	AYGA	-6.081689	145.391881	5282	10.0	U
1	2	Madang	Madang	Papua New Guinea	MAG	AYMD	-5.207083	145.788700	20	10.0	U
2	3	Mount Hagen	Mount Hagen	Papua New Guinea	HGU	AYMH	-5.826789	144.295861	5388	10.0	U

Question 1.1 (12% credit)

It looks like the departure and arrival in `flights` were read in as floating-point numbers. Write two functions, `extract_hour` and `extract_mins` that converts military time to hours and minutes, respectively. Hint: You may want to use modular arithmetic and integer division. Keep in mind that the data has not been cleaned and you need to check whether the extracted values are valid. Replace all the invalid values with `NaN`. The documentation for `pandas.Series.where` provided here (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.where.html) should be helpful.

In [219]:

```
# 5% credit
def extract_hour(time):
    """
    Extracts hour information from military time.

    Args:
        time (float64): series of time given in military format.
          Takes on values in 0.0-2359.0 due to float64 representation.

    Returns:
        array (float64): series of input dimension with hour information.
          Should only take on integer values in 0-23
    """
    corrected = time.where(time.notna(), np.nan)
    result = []

    for time_val in corrected:
        final_val = None
        # 1.0 12.0 230.0 1430.0
        # one digit -> hour
        if time_val <= 9.0 and time_val >= 0:
            final_val = time_val
        # two digits -> hours
        if time_val < 24.0 and time_val >= 10.0 and final_val == None:
            final_val = time_val

        # three digits
        if time_val >= 100.0 and time_val <= 959.0 and final_val == None:
            final_val = time_val // 100

        # 4 digits
        if time_val >= 1000.0 and time_val <= 2359:
             final_val = time_val // 100

        if time_val == None:
            final_val = np.nan

        result.append(final_val)

    return pd.Series(result, dtype='float64')
```

In [123]:

```
# 1% credit
### write code to test your extract_hour function here and execute it
# HINT: See tests_sample_part1/tests.py
extract_hour(flights['sched_dep_time'])
```

Out[123]:

```
0         6.0
1         6.0
2         6.0
3         6.0
4         6.0
         ...
16856    12.0
16857    11.0
16858    17.0
16859    20.0
16860    14.0
Length: 16861, dtype: float64
```

In [218]:

```
# 5% credit
def extract_mins(time):
    """
    Extracts minute information from military time

    Args:
        time (float64): series of time given in military format.
          Takes on values in 0.0-2359.0 due to float64 representation.

    Returns:
        array (float64): series of input dimension with minute information.
          Should only take on integer values in 0-59
    """
    corrected = time.where(time.notna(), np.nan)
    result = []

    for time_val in corrected:
        final_val = None
        # 1.0 12.0 230.0 1430.0
        # one digit -> 0 minutes
        if time_val <= 9.0 and time_val >= 0:
            final_val = 0.0
        # two digits -> 0 minutes
        if time_val < 24.0 and time_val >= 10.0 and final_val == None:
            final_val = 0.0

        # three digits -> last is minute val
        if time_val >= 100.0 and time_val <= 959.0 and final_val == None:
            final_val = time_val % 100

        # 4 digits -> last two are minutes
        if time_val >= 1000.0 and time_val <= 2359:
             final_val = time_val % 100

        if time_val == None:
            final_val = np.nan

        result.append(final_val)

    return pd.Series(result, dtype='float64')
```

In [125]:

```
# 1% credit
### write code to test your extract_mins function here and execute it
# HINT: See tests_sample_part1/tests.py
extract_mins(flights['sched_dep_time'])
```

Out[125]:

```
0        30.0
1         0.0
2        30.0
3        30.0
4        40.0
         ... 
16856    45.0
16857    30.0
16858    40.0
16859     0.0
16860    30.0
Length: 16861, dtype: float64
```

Question 1.2 (13% credit)

Using your two functions above, filter the `flights` data for flights that departed 20 or more minutes later than scheduled by comparing `sched_dep_time` and `actual_dep_time`. You need not worry about flights that were delayed to the next day for this question.

In [216]:

```
# 5% credit
def convert_to_minofday(time):
    """
    Converts military time to minute of day

    Args:
        time (float64): series of time given in military format.
          Takes on values in 0.0-2359.0 due to float64 representation.

    Returns:
        array (float64): series of input dimension with minute of day

    Example: 1:03pm is converted to 783.0
    """
    result = []
    hours_frame = extract_hour(time)
    minutes_frame = extract_mins(time)

    # go through each of them in parallel and add up
    for i in range(len(hours_frame)):
        result.append(hours_frame[i]*60.0 + minutes_frame[i])

    return pd.Series(result, dtype="float64")

# Test your code
ser = pd.Series([1303, 1200, 2400], dtype='float64')
convert_to_minofday(ser)
# 0    783.0
# 1    720.0
# 2      NaN
# dtype: float64
```

Out[216]:

```
0    783.0
1    720.0
2      NaN
dtype: float64
```

In [215]:

```
# 5% credit
def calc_time_diff(x, y):
    """
    Calculates delay times y - x

    Args:
        x (float64): series of scheduled time given in military format.
          Takes on values in 0.0-2359.0 due to float64 representation.
        y (float64): series of same dimensions giving actual time

    Returns:
        array (float64): series of input dimension with delay time
    """
    result = []

    scheduled = convert_to_minofday(x)
    actual = convert_to_minofday(y)

    for i in range(len(actual)):
        result.append(actual[i] - scheduled[i])

    return pd.Series(result, dtype="float64")

#Test your code
sched = pd.Series([1303, 1210], dtype='float64')
actual = pd.Series([1304, 1215], dtype='float64')
calc_time_diff(sched, actual)
# 0    1.0
# 1    5.0
# dtype: float64
```

Out[215]:

```
0    1.0
1    5.0
dtype: float64
```

In [213]:

```
# 3% credit
### write code to test your functions here by calculating delay between `sc
### your printed results should show the values of the following two variab

# Series object showing delay time
delay = calc_time_diff(flights['sched_dep_time'], flights['actual_dep_time'

# Dataframe showing flights delayed by 20 minutes or more
delayed20 = []
for i in range(len(delay)):
    if delay[i] >= 20.0:
        delayed20.append(flights.loc[i])
delayed20
```

Out[213]:

```
[year                      2016
 month                        1
 day                         16
 date                2016-01-16
 carrier                     AA
 tailnum                 N3GAAA
 flight                     208
 origin                     SFO
 destination                MIA
 sched_dep_time           640.0
 actual_dep_time          723.0
 sched_arr_time          1458.0
 actual_arr_time         1534.0
 Name: 15, dtype: object,
 year                      2016
 month                        1
 day                         20
 date                2016-01-20
 carrier                     AA
```

Question 1.3 (5% credit)

Using your answer from question 1.2, find the full name of every destination city with a flight from SFO or OAK that was delayed by 20 or more minutes. The airport codes used in `flights` are IATA codes. Sort the cities alphabetically. Make sure you remove duplicates. You may find `drop_duplicates` and `sort_values` helpful.

In [87]:

```
# 5% credit
### your printed results should show the values of the following two variab
# HINT: You will need to use `delayed20` and `airport` dataframes
delayed_airports = [] # Dataframe showing airports that satisfy above condi
# delayed_destinations = ... # Unique and sorted destination cities

# I don't kow how to do it.
```

Part 2 (50% of HW 1): Web scraping and data collection

Here, you will practice collecting and processing data in Python. By the end of this exercise hopefully you should look at the wonderful world wide web without fear, comforted by the fact that anything you can see with your human eyes, a computer can see with its computer eyes. In particular, we aim to give you some familiarity with:

- Using HTTP to fetch the content of a website
- HTTP Requests (and lifecycle)
- RESTful APIs
  - Authentication (OAuth)
  - Pagination
  - Rate limiting
- JSON vs. HTML (and how to parse each)
- HTML traversal (CSS selectors)

Since everyone loves food (presumably), the ultimate end goal of this homework will be to acquire the data to answer some questions and hypotheses about the restaurant scene in Chicago (which we will get to later). We will download **both** the metadata on restaurants in Chicago from the Yelp API and with this metadata, retrieve the comments/reviews and ratings from users on restaurants.

Library Documentation

For solving this part, you need to look up online documentation for the Python packages you will use:

- Standard Library:
  - io (https://docs.python.org/3/library/io.html)
  - time (https://docs.python.org/3/library/time.html)
  - json (https://docs.python.org/3/library/json.html)
- Third Party
  - requests (http://docs.python-requests.org/en/master/)
  - Beautiful Soup (version 4) (https://www.crummy.com/software/BeautifulSoup/bs4/doc/)
  - yelp-fusion (https://www.yelp.com/developers/documentation/v3/get_started)

Note: You may come across a `yelp-python` library online. The library is deprecated and incompatible with the current Yelp API, so do not use the library.

Setup

First, import necessary libraries:

In [119]:

```
import io, time, json
import requests
from bs4 import BeautifulSoup
```

Authentication and working with APIs

There are various authentication schemes that APIs use, listed here in relative order of complexity:

- No authentication

- HTTP basic authentication (https://en.wikipedia.org/wiki/Basic_access_authentication)
- Cookie based user login
- OAuth (v1.0 & v2.0, see this post (http://stackoverflow.com/questions/4113934/how-is-oauth-2-different-from-oauth-1) explaining the differences)
- API keys
- Custom Authentication

For the NYT example below (**Q2.1**), since it is a publicly visible page we did not need to authenticate. HTTP basic authentication isn't too common for consumer sites/applications that have the concept of user accounts (like Facebook, LinkedIn, Twitter, etc.) but is simple to setup quickly and you often encounter it on with individual password protected pages/sites.

Cookie based user login is what the majority of services use when you login with a browser (i.e. username and password). Once you sign in to a service like Facebook, the response stores a cookie in your browser to remember that you have logged in (HTTP is stateless). Each subsequent request to the same domain (i.e. any page on `facebook.com`) also sends the cookie that contains the authentication information to remind Facebook's servers that you have already logged in.

Many REST APIs however use OAuth (authentication using tokens) which can be thought of a programmatic way to "login" *another* user. Using tokens, a user (or application) only needs to send the login credentials once in the initial authentication and as a response from the server gets a special signed token. This signed token is then sent in future requests to the server (in place of the user credentials).

A similar concept common used by many APIs is to assign API Keys to each client that needs access to server resources. The client must then pass the API Key along with *every* request it makes to the API to authenticate. This is because the server is typically relatively stateless and does not maintain a session between subsequent calls from the same client. Most APIs (including Yelp) allow you to pass the API Key via a special HTTP Header: `Authorization: Bearer <API_KEY>` . Check out the docs (https://www.yelp.com/developers/documentation/v3/authentication) for more information.

Question 2.1: Basic HTTP Requests w/o authentication (6%)

First, let's do the "hello world" of making web requests with Python to get a sense for how to programmatically access web pages: an (unauthenticated) HTTP GET to download a web page.

Fill in the funtion to use `requests` to download and return the raw HTML content of the URL passed in as an argument. As an example try the following NYT article (on Youtube's algorithmic recommendation): https://www.nytimes.com/2019/03/29/technology/youtube-online-extremism.html (https://www.nytimes.com/2019/03/29/technology/youtube-online-extremism.html)

Your function should return a tuple of: (`<status_code>` , `<text>`). (Hint: look at the **Library documentation** listed earlier to see how `requests` should work.)

In [211]:
```
# 3% credit
def retrieve_html(url):
    """
    Return the raw HTML at the specified URL.

    Args:
        url (string):

    Returns:
        status_code (integer):
        raw_html (string): the raw HTML content of the response, properly e
    """
    ret_tuple = (requests.get(url).status_code, requests.get(url).text)

    return ret_tuple
```

In [212]:
```
# 3% credit
youtube_article = retrieve_html('https://www.nytimes.com/2019/03/29/technol
print(youtube_article)
# (200, '<!DOCTYPE html>\n<html lang="en" class="story" xmlns:og="http://op
```

```
(200, '<!DOCTYPE html>\n<html lang="en" class="story nytapp-vi-article"
xmlns:og="http://opengraphprotocol.org/schema/">\n  <head>\n    <meta cha
rset="utf-8" />\n    <title data-rh="true">YouTube’s Product Chief on Onl
ine Radicalization and Algorithmic Rabbit Holes - The New York Times</tit
le>\n    <meta data-rh="true" name="robots" content="noarchive, max-image
-preview:large"/><meta data-rh="true" name="description" content="Neal Mo
han discusses the streaming site’s recommendation engine, which has becom
e a growing liability amid accusations that it steers users to increasing
ly extreme content."/><meta data-rh="true" property="og:url" content="htt
ps://www.nytimes.com/2019/03/29/technology/youtube-online-extremism.htm
l"/><meta data-rh="true" property="og:type" content="article"/><meta data
-rh="true" property="og:title" content="YouTube’s Product Chief on Online
Radicalization and Algorithmic Rabbit Holes (Published 2019)"/><meta data
-rh="true" property="og:image" content="https://static01.nyt.com/images/2
019/03/29/business/29roose-1/29roose-1-facebookJumbo.jpg?year=2019&amp;h=
549&amp;w=1050&amp;s=ae1f74fcc17415f17e1ff61b3119d6454967d7b7eb91fbfd22c2
d42aa51bdf91&amp;k=ZQJBKqZ0VN"/><meta data-rh="true" property="og:image:a
lt" content="Neal Mohan is YouTube’s chief product officer."/><meta data-
rh="true" property="og:description" content="Neal Mohan discusses the str
eaming site’s recommendation engine, which has become a growing liability
```

Now while this example might have been fun, we haven't yet done anything more than we could with a web browser. To really see the power of programmatically making web requests we will need to interact with an API. For the rest of this lab we will be working with the Yelp API (https://www.yelp.com/developers/documentation/v3/get_started) and Yelp data (for an extensive data dump see their Academic Dataset Challenge (https://www.yelp.com/dataset_challenge)).

Yelp API Access

The reasons for using the Yelp API are 3 fold:

1. Incredibly rich dataset that combines:
    - entity data (users and businesses)

- preferences (i.e. ratings)
- geographic data (business location and check-ins)
- temporal data
- text in the form of reviews
- and even images.

2. Well documented API (https://www.yelp.com/developers/documentation/v3/get_started) with thorough examples.
3. Extensive data coverage so that you can find data that you know personally (from your home town/city or account). This will help with understanding and interpreting your results.

Yelp used to use OAuth tokens but has now switched to API Keys. **For the sake of backwards compatibility Yelp still provides a Client ID and Secret for OAuth, but you will not need those for this assignment.**

To access the Yelp API, we will need to go through a few more steps than we did with the first NYT example. Most large web scale companies use a combination of authentication and rate limiting to control access to their data to ensure that everyone using it abides. The first step (even before we make any request) is to setup a Yelp account if you do not have one and get API credentials.

1. Create a Yelp (https://www.yelp.com/login) account (if you do not have one already)
2. Generate API keys (https://www.yelp.com/developers/v3/manage_app) (if you haven't already). You will only need the API Key (not the Client ID or Client Secret) -- more on that later.

Now that we have our accounts setup we can start making requests!

Question 2.2: Authenticated HTTP Request with the Yelp API (16%)

First, store your Yelp credentials in a local file (kept out of version control) which you can read in to authenticate with the API. This file can be any format/structure since you will fill in the function stub below.

For example, you may want to store your key in a file called `yelp_api_key.txt` (run in terminal):

```
echo 'YOUR_YELP_API_KEY' > yelp_api_key.txt
```

KEEP THE API KEY FILE PRIVATE AND OUT OF VERSION CONTROL (and definitely do not submit them to Gradescope!)

You can then read from the file using:

In [100]:
```
# 3% credit
with open('yelp_api_key.txt', 'r') as f:
    api_key = f.read().replace('\n','')
    print(api_key)
    # verify your api_key is correct
# DO NOT FORGET TO CLEAR THE OUTPUT TO KEEP YOUR API KEY PRIVATE
```

```
UaEvEI1ip95yv7yiFv1MgIgRRPvGe4D0cUFUcmcKLIE-OV97kEdakKZoQvfWRbN7YP_HQE4U_
C8lZLbEFBRlBSemIbEx1KhNxUivMqvNJTfO5U4OUd_7mOV0DCn_YXYx
```

In [101]:

```
# 3% credit
def read_api_key(filepath):
    """
    Read the Yelp API Key from file.
    
    Args:
        filepath (string): File containing API Key
    Returns:
        api_key (string): The API Key
    """
    
    # feel free to modify this function if you are storing the API Key diff
    with open(filepath, 'r') as f:
        return f.read().replace('\n','')
```

Using the Yelp API, fill in the following function stub to make an authenticated request to the search (https://www.yelp.com/developers/documentation/v3/business_search) endpoint. Remember Yelp allows you to pass the API Key via a special HTTP Header: `Authorization: Bearer <API_KEY>` . Check out the docs (https://www.yelp.com/developers/documentation/v3/authentication) for more information.

In [155]:

```
# 4% credit
def location_search_params(api_key, location, **kwargs):
    """
    Construct url, headers and url_params. Reference API docs (link above)
    """
    url = 'https://api.yelp.com/v3/businesses/search'
    auth = "Bearer " + api_key
    headers = {"Authorization": auth}
    # SPACES in url is problematic. How should you handle location containi
    location = location.strip()
    location = location.replace(" ", "+")
    url_params = {"location": location, **kwargs}
    
    return url, headers, url_params
```

Hint: `**kwargs` represent keyword arguments that are passed to the function. For example, if you called the function `location_search_params(api_key, location, offset=0, limit=50)` . The arguments `api_key` and `location` are called *positional arguments* and key-value pair arguments are called **keyword arguments**. Your `kwargs` variable will be a python dictionary with those keyword arguments.

In [158]:

```
# Test your code
api_key = "test_api_key_xyz"
location = "Chicago"
url, headers, url_params = location_search_params(api_key, location, offset
url, headers, url_params
# ('https://<hidden_url_check_search_endpoint_docs_to_get_answer>',
#  {'Authorization': 'Bearer test_api_key_xyz'},
#  {'location': 'Chicago', 'offset': 0, 'limit': 50})
```

Out[158]:

```
('https://api.yelp.com/v3/businesses/search',
 {'Authorization': 'Bearer test_api_key_xyz'},
 {'location': 'Chicago', 'offset': 0, 'limit': 50})
```

Now use `location_search_params(api_key, location, **kwargs)` to actually search restaurants from Yelp API. Most of the code is provided to you. Complete the `api_get_request` function given below.

In [159]:

```
# 3% credit
def api_get_request(url, headers, url_params):
    """
    Send a HTTP GET request and return a json response

    Args:
        url (string): API endpoint url
        headers (dict): A python dictionary containing HTTP headers includi
        url_params (dict): The parameters (required and optional) supported

    Returns:
        results (json): response as json
    """
    http_method = 'GET'
    # See requests.request?
    response = requests.get(url, headers=headers, params=url_params).json()
    return response


def yelp_search(api_key, location, offset=0):
    """
    Make an authenticated request to the Yelp API.

    Args:
        api_key (string): Your Yelp API Key for Authentication
        location (string): Business Location
        offset (int): param for pagination

    Returns:
        total (integer): total number of businesses on Yelp corresponding t
        businesses (list): list of dicts representing each business
    """
    url, headers, url_params = location_search_params(api_key, location, of
    response_json = api_get_request(url, headers, url_params)
    return response_json["total"], list(response_json["businesses"])

#3% credit
api_key = read_api_key('yelp_api_key.txt')
num_records, data = yelp_search(api_key, 'Chicago')
print(num_records)
#240
print(len(data))
#20
print(list(map(lambda x: x['name'], data)))
#['Girl & The Goat', 'Wildberry Pancakes and Cafe', 'Au Cheval', 'The Purpl
```

```
8600
20
['Girl & The Goat', 'Wildberry Pancakes and Cafe', 'Au Cheval', 'The Purp
le Pig', "Lou Malnati's Pizzeria", 'Art Institute of Chicago', "Bavette's
Bar & Boeuf", 'Cafe Ba-Ba-Reeba!', 'Smoque BBQ', 'Little Goat Diner', "Pe
quod's Pizzeria", 'Quartino Ristorante', 'Alinea', "Kuma's Corner - Belmo
nt", "Joe's Seafood, Prime Steak & Stone Crab", 'Crisp', "Portillo's Hot
Dogs", 'Sapori Trattoria', 'Xoco', "Molly's Cupcakes"]
```

Now that we have completed the "hello world" of working with the Yelp API, we are ready to really fly! The rest of the exercise will have a bit less direction since there are a variety of ways to retrieve the requested information but you should have all the component knowledge at this point to work with the API. Yelp being a fairly general platform actually has many more business than just restaurants, but by using the flexibility of the API we can ask it to only return the restaurants.

Parameterization and Pagination

And before we can get any reviews on restaurants, we need to actually get the metadata on ALL of the restaurants in Chicago. Notice above that while Yelp told us that there are ~240, the response contained fewer actual `Business` objects. This is due to pagination and is a safeguard against returning **TOO** much data in a single request (what would happen if there were 100,000 restaurants?) and can be used in conjuction with *rate limiting* as well as a way to throttle and protect access to Yelp data.

> As a thought exercise, consider: If an API has 1,000,000 records, but only returns 10 records per page and limits you to 5 requests per second... how long will it take to acquire ALL of the records contained in the API?

One of the ways that APIs are an improvement over plain web scraping is the ability to make **parameterized** requests. Just like the Python functions you have been writing have arguments (or parameters) that allow you to customize its behavior/actions (an output) without having to rewrite the function entirely, we can parameterize the queries we make to the Yelp API to filter the results it returns.

Question 2.3: Acquire all of the restaurants in Chicago on Yelp (10%)

Again using the [API documentation (https://www.yelp.com/developers/documentation/v3/business_search)](https://www.yelp.com/developers/documentation/v3/business_search) for the `search` endpoint, fill in the following function to retrieve all of the *Restuarants* (using categories) for a given query. Again you should use your `read_api_key()` function outside of the `all_restaurants()` stub to read the API Key used for the requests. You will need to account for **pagination** and **[rate limiting (https://www.yelp.com/developers/faq)](https://www.yelp.com/developers/faq)** to:

1. Retrieve all of the Business objects (# of business objects should equal `total` in the response). **Paginate by querying 10 restaurants each request.**
2. Pause slightly (at least 200 milliseconds) between subsequent requests so as to not overwhelm the API (and get blocked).

As always with API access, make sure you follow all of the [API's policies (https://www.yelp.com/developers/api_terms)](https://www.yelp.com/developers/api_terms) and use the API responsibly and respectfully.

DO NOT MAKE TOO MANY REQUESTS TOO QUICKLY OR YOUR KEY MAY BE BLOCKED

In [161]:

```
# 4% credit
import math


def paginated_restaurant_search_requests(api_key, location, total):
    """
    Returns a list of tuples (url, headers, url_params) for paginated searc
    Args:
        api_key (string): Your Yelp API Key for Authentication
        location (string): Business Location
        total (int): Total number of items to be fetched
    Returns:
        results (list): list of tuple (url, headers, url_params)
    """
    # HINT: Use total, offset and limit for pagination
    # You can reuse function location_search_params(...)
    all_requests_data = []

    for i in range(math.ceil(total/10)):
        generated_req_data = location_search_params(api_key, location, offs
        all_requests_data.append(generated_req_data)

    return all_requests_data

# Test your code
api_key = read_api_key('yelp_api_key.txt')
location = "Chicago"
all_restaurants_requests = paginated_restaurant_search_requests(api_key, lo
all_restaurants_requests

# [('https:<hidden>',
#   {'Authorization': 'Bearer test_api_key_xyz'},
#   {'location': 'Chicago',
#    'offset': 0,
#    'limit': 10,
#    'categories': '<hidden>'}),
#  ('https:<hidden>',
#   {'Authorization': 'Bearer test_api_key_xyz'},
#   {'location': 'Chicago',
#    'offset': 10,
#    'limit': 10,
#    'categories': '<hidden>'})]
```

Out[161]:

```
[('https://api.yelp.com/v3/businesses/search',
  {'Authorization': 'Bearer UaEvEI1ip95yv7yiFv1MgIgRRPvGe4D0cUFUcmcKLIE-O
V97kEdakKZoQvfWRbN7YP_HQE4U_C8lZLbEFBRlBSemIbEx1KhNxUivMqvNJTfO5U4OUd_7mO
V0DCn_YXYx'},
  {'location': 'Chicago',
   'offset': 0,
   'limit': 10,
   'categories': 'restaurants'}),
 ('https://api.yelp.com/v3/businesses/search',
  {'Authorization': 'Bearer UaEvEI1ip95yv7yiFv1MgIgRRPvGe4D0cUFUcmcKLIE-O
V97kEdakKZoQvfWRbN7YP_HQE4U_C8lZLbEFBRlBSemIbEx1KhNxUivMqvNJTfO5U4OUd_7mO
V0DCn_YXYx'},
  {'location': 'Chicago',
   'offset': 10,
   'limit': 10,
   'categories': 'restaurants'})]
```

In [184]:

```
# 3% credit
def all_restaurants(api_key, location):
    """
    Construct the pagination requests for ALL the restaurants on Yelp for a

    Args:
        api_key (string): Your Yelp API Key for Authentication
        location (string): Business Location

    Returns:
        results (list): list of dicts representing each restaurant
    """
    all_responses = []
    # What keyword arguments should you pass to get first page of restauran
    url, headers, url_params = location_search_params(api_key, location, li
    response_json = api_get_request(url, headers, url_params)
    total_items = response_json["total"]

    all_restaurants_requests = paginated_restaurant_search_requests(api_key

    # Use returned list of (url, headers, url_params) and function api_get_
    # REMEMBER to pause slightly after each request.
    for i in range(len(all_restaurants_requests)):
        url, headers, url_params = all_restaurants_requests[i]
        time.sleep(0.25)
        resp = requests.get(url, headers=headers, params=url_params).json()
        # array extends the other array
        all_responses += resp['businesses']


    return all_responses
```

You can test your function with an individual neighborhood in Chicago (for example, Greektown). Chicago itself has a lot of restaurants... meaning it will take a lot of time to download them all.

```
In [186]: # 3% credit
          api_key = read_api_key('yelp_api_key.txt')
          data = all_restaurants(api_key, 'Greektown, Chicago, IL')
          print(len(data))
          # 99
          print(list(map(lambda x:x['name'], data)))
          # ['Greek Islands Restaurant', 'Artopolis', 'Meli Cafe & Juice Bar', 'Athen
```

```
96
['Greek Islands Restaurant', 'Artopolis', 'Meli Cafe & Juice Bar', 'Athen
a Greek Restaurant', 'WJ Noodles', 'Zeus Restaurant', 'Green Street Smoke
d Meats', 'Mr Greek Gyros', "Philly's Best", 'Monteverde', 'Primos Chicag
o Pizza Pasta', 'J.P. Graziano Grocery', '9 Muses', 'Green Street Local',
'Sepia', 'High Five Ramen', 'Spectrum Bar and Grill', 'Dawali Jerusalem K
itchen', "Lou Mitchell's", "Nando's PERi-PERi", "Formento's", 'Xi’an Cuis
ine', 'Jubilee Juice & Grill', 'Taco Burrito King - Greektown', 'H Mart -
Chicago', 'Parlor Pizza Bar', 'Omakase Yume', 'The Madison Bar & Kitche
n', 'Blaze Pizza', 'Booze Box', 'El Che Steakhouse & Bar', 'Trivoli Taver
n', 'M2 Cafe', 'Yolk West Loop', 'Bandit', "Nonna's Pizza & Sandwiches",
'Morgan Street Cafe', "Giordano's", 'Veros Caffe and Gelato', 'Ciao! Cafe
& Wine Lounge', 'Rye Deli & Drink', 'Umami Burger - West Loop', "Nancy's
Pizza", 'Slightly Toasted', 'Sushi Pink', 'Aroma Desi Grill', 'Epic Burge
r', 'SGD Dubu So Gong Dong Tofu & Korean BBQ', 'Taco Lulú', "Hannah's Bre
tzel", 'Beggars Pizza', 'TenGoku Aburiya', "Jet's Pizza", 'Naf Naf Gril
l', 'Asadito', "Wok N' Bao", 'I Dream of Falafel', 'Stelios Bottles & Bit
es', 'Oki Sushi', 'Pockets', "Jimmy John's", 'Klay Oven Kitchen', "Cafe
L'ami", 'K-Kitchen', "Sang's Kitchen", 'Freshii', 'Subway', "Bebe's Koshe
r Deli", 'Roti Modern Mediterranean', 'Chipotle Mexican Grill', "JoKeR's
Cajun Kitchen", 'Baci Amore', 'Corner Bakery', 'Potbelly Sandwich Shop',
'The Ruin Daily', 'Five Guys', 'Izakaya yume', 'Potbelly Sandwich Shop',
'Krispy Rice', "Domino's Pizza", 'Downstate Donuts', 'Taco Bell Cantina',
'Red Star Bar', "Jimmy John's", 'Great Steak', 'Panera Bread', 'Burger Ki
ng', 'Paper Thin Pizza', 'Hunan House', 'this little goat kitchen', 'Spak
eteria', 'Flik International', "Harold's Chicken On Clinton", "Sam's Cris
py Chicken - West Loop", 'Cafe Italo', 'Subway']
```

Now that we have the metadata on all of the restaurants in Greektown (or at least the ones listed on Yelp), we can retrieve the reviews and ratings. The Yelp API gives us aggregate information on ratings but it doesn't give us the review text or individual users' ratings for a restaurant. For that we need to turn to web scraping, but to find out what pages to scrape we first need to parse our JSON from the API to extract the URLs of the restaurants.

In general, it is a best practice to separate the act of **downloading** data and **parsing** data. This ensures that your data processing pipeline is modular and extensible (and autogradable ;). This decoupling also solves the problem of expensive downloading but cheap parsing (in terms of computation and time).

Question 2.4: Parse the API Responses and Extract the URLs (7%)

Because we want to separate the **downloading** from the **parsing**, fill in the following function to parse the URLs pointing to the restaurants on `yelp.com` . As input your function should expect a string of properly formatted JSON (http://www.json.org/) (which is similar to **BUT** not the same as a Python dictionary) and as output should return a Python list of strings. Hint: print your `data` to see the JSON-formatted information you have. The input JSON will be structured as follows (same as the sample (https://www.yelp.com/developers/documentation/v3/business_search) on the Yelp API page):

```
{
  "total": 8228,
  "businesses": [
    {
      "rating": 4,
      "price": "$",
      "phone": "+14152520800",
      "id": "four-barrel-coffee-san-francisco",
      "is_closed": false,
      "categories": [
        {
          "alias": "coffee",
          "title": "Coffee & Tea"
        }
      ],
      "review_count": 1738,
      "name": "Four Barrel Coffee",
      "url": "https://www.yelp.com/biz/four-barrel-coffee-san-francisco",
      "coordinates": {
        "latitude": 37.7670169511878,
        "longitude": -122.42184275
      },
      "image_url": "http://s3-media2.fl.yelpcdn.com/bphoto/MmgtASP3l_t4tPCL1iAsCg/o.jpg",
      "location": {
        "city": "San Francisco",
        "country": "US",
        "address2": "",
        "address3": "",
        "state": "CA",
        "address1": "375 Valencia St",
        "zip_code": "94103"
      },
      "distance": 1604.23,
      "transactions": ["pickup", "delivery"]
    }
  ],
  "region": {
    "center": {
      "latitude": 37.767413217936834,
      "longitude": -122.42820739746094
    }
  }
}
```

In [193]:

```
# 4% credit
def parse_api_response(data):
    """
    Parse Yelp API results to extract restaurant URLs.

    Args:
        data (string): String of properly formatted JSON.

    Returns:
        (list): list of URLs as strings from the input JSON.
    """

    json_arr = json.loads(data)

    urls = []

    for el in json_arr:
        urls.append(el['url'])

    return urls


# 3% credit
url, headers, url_params = location_search_params(api_key, "Bridgeport, Chi
response_text = api_get_request(url, headers, url_params)
parse_api_response(json.dumps(response_text['businesses']))
# ['https://www.yelp.com/biz/nana-chicago?adjust_creative=ioqEYAcUhZO272qCI
#  'https://www.yelp.com/biz/bridgeport-coffee-chicago-4?adjust_creative=io
# ...]
```

Out[193]:

```
['https://www.yelp.com/biz/nana-chicago?adjust_creative=QEUJPh6L9o3Lhkd32
1XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_sour
ce=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/jackalope-coffee-and-tea-house-chicago?adjust_
creative=QEUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v
3_business_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/marias-packaged-goods-and-community-bar-chicag
o?adjust_creative=QEUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_med
ium=api_v3_business_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/bridgeport-coffee-chicago-4?adjust_creative=QE
UJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_
search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/mins-noodle-house-%E6%B8%94%E5%AE%B6%E9%87%8D%
E5%BA%86%E5%B0%8F%E9%9D%A2-chicago-32?adjust_creative=QEUJPh6L9o3Lhkd321X
nJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_source
=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/the-duck-inn-chicago?adjust_creative=QEUJPh6L9
o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&
utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/francos-ristorante-chicago?adjust_creative=QEU
JPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_s
earch&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/zaytune-mediterranean-grill-chicago-4?adjust_c
reative=QEUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3
_business_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/gios-cafe-and-deli-chicago?adjust_creative=QEU
JPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_s
earch&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/han-202-chicago?adjust_creative=QEUJPh6L9o3Lhk
d321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_s
ource=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/phils-pizza-chicago?adjust_creative=QEUJPh6L9o
3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&u
tm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/potsticker-house-chicago?adjust_creative=QEUJP
h6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_sea
rch&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/stix-n-brix-wood-fired-pizza-chicago?adjust_cr
eative=QEUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_
business_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/bernices-tavern-chicago?adjust_creative=QEUJPh
6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_sear
ch&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/taipei-cafe-chicago?adjust_creative=QEUJPh6L9o
3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&u
tm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/bridgeport-bakery-2-0-chicago?adjust_creative=
QEUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_busines
s_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/pancho-pistolas-chicago?adjust_creative=QEUJPh
6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_sear
ch&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/south-kawa-chicago?adjust_creative=QEUJPh6L9o3
Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&ut
m_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/shinya-ramen-house-chicago-3?adjust_creative=Q
EUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business
_search&utm_source=QEUJPh6L9o3Lhkd321XnJA',
 'https://www.yelp.com/biz/pleasant-house-pub-chicago-3?adjust_creative=Q
EUJPh6L9o3Lhkd321XnJA&utm_campaign=yelp_api_v3&utm_medium=api_v3_business
_search&utm_source=QEUJPh6L9o3Lhkd321XnJA']
```

As we can see, JSON is quite trivial to parse (which is not the case with HTML as we will see in a second) and work with programmatically. This is why it is one of the most ubiquitous data serialization formats (especially for ReSTful APIs) and a huge benefit of working with a well defined API if one exists. But APIs do not always exists or provide the data we might need, and as a last resort we can always scrape web pages...

Working with Web Pages (and HTML)

Think of APIs as similar to accessing an application's database itself (something you can interactively query and receive structured data back). But the results are usually in a somewhat raw form with no formatting or visual representation (like the results from a database query). This is a benefit *AND* a drawback depending on the end use case. For data science and *programatic* analysis this raw form is quite ideal, but for an end user requesting information from a *graphical interface* (like a web browser) this is very far from ideal since it takes some cognitive overhead to interpret the raw information. And vice versa, if we have HTML it is quite easy for a human to visually interpret it, but to try to perform some type of programmatic analysis we first need to parse the HTML into a more structured form.

> As a general rule of thumb, if the data you need can be accessed or retrieved in a structured form (either from a bulk download or API) prefer that first. But if the data you want (and need) is not as in our case we need to resort to alternative (messier) means.

Going back to the "hello world" example of question 2.1 with the NYT, we will do something similar to retrieve the HTML of the Yelp site itself (rather than going through the API programmatically) as text.

> However, we will use saved HTML pages to reduce excessive traffic to the Yelp website.

Question 2.5: Parse a Yelp restaurant Page (4%)

Using `BeautifulSoup` , parse the HTML of a single Yelp restaurant page to extract the reviews in a structured form as well as the URL to the next page of reviews (or `None` if it is the last page). Fill in following function stubs to parse a single page of reviews and return:

- the reviews as a structured Python dictionary
- the HTML element containing the link/url for the next page of reviews (or None).

For each review be sure to structure your Python dictionary as follows (to be graded correctly). The order of the keys doesn't matter, only the keys and the data type of the values:

```
{
    'author': str
    'rating': float
    'date': str ('yyyy-mm-dd')
    'description': str
}

# Example
{
    'author': 'Topsy Kretts'
    'rating': 4.7
    'date': '2016-01-23'
    'description': "Wonderful!"
}
```

There can be issues with Beautiful Soup using various parsers, for maximum compatibility (and fewest errors) initialize the library with the default (and Python standard library parser): `BeautifulSoup(markup, "html.parser")` .

Most of the function has been provided to you:

In [202]:

```
# 4% credit
url_lookup = {
"https://www.yelp.com/biz/the-jibarito-stop-chicago-2?start=225":"parse_pag
"https://www.yelp.com/biz/the-jibarito-stop-chicago-2?start=245":"parse_pag
}

def html_fetcher(url):
    """
    Return the raw HTML at the specified URL.
    Args:
        url (string):

    Returns:
        status_code (integer):
        raw_html (string): the raw HTML content of the response, properly e
    """
    html_file = url_lookup.get(url)
    with open(html_file, 'rb') as file:
        html_text = file.read()
        return 200, html_text


def parse_page(html):
    """
    Parse the reviews on a single page of a restaurant.

    Args:
        html (string): String of HTML corresponding to a Yelp restaurant

    Returns:
        tuple(list, string): a tuple of two elements
            first element: list of dictionaries corresponding to the extrac
            second element: URL for the next page of reviews (or None if it
    """
    soup = BeautifulSoup(html,'html.parser')
    url_next = soup.find('link',rel='next')
    if url_next:
        url_next = url_next.get('href')
    else:
        url_next = None

    reviews = soup.find_all('div', itemprop="review")

    result = []
    found_data = {
        "authors_arr": None,
        "ratings_arr": None,
        "dates_arr": None,
        "desc_arr": None
    }
    # find all meta data
    found_data['ratings_arr'] = soup.find_all('meta', itemprop="ratingValue
    found_data['authors_arr'] = soup.find_all('meta', itemprop="author")
    found_data['desc_arr'] = soup.find_all('p', itemprop="description")
    found_data['dates_arr'] = soup.find_all('meta', itemprop="datePublished
    # HINT: print reviews to see what http tag to extract
    for i in range(len(reviews)):
        # get vals
        author_val =  found_data['authors_arr'][i]['content']
        rating_val =  found_data['ratings_arr'][i]['content']
        date_val = found_data['dates_arr'][i]['content']
        desc_val = found_data['desc_arr'][i].get_text()
        # form a result entity
        result.append({
            'author': author_val,
            'rating': float(rating_val),
            'date': date_val,
            'description': desc_val
        })

    return result, url_next

# Test your implementation
code, html = html_fetcher("https://www.yelp.com/biz/the-jibarito-stop-chica
reviews_list, url_next = parse_page(html)
print(len(reviews_list)) # 20
print(url_next) #https://www.yelp.com/biz/the-jibarito-stop-chicago-2?start
```

```
20
https://www.yelp.com/biz/the-jibarito-stop-chicago-2?start=245 (https://w
ww.yelp.com/biz/the-jibarito-stop-chicago-2?start=245)
```

Question 2.6: Extract all Yelp reviews for a Single Restaurant (7%)

So now that we have parsed a single page, and figured out a method to go from one page to the next we are ready to combine these two techniques and actually crawl through web pages!

Using the provided `html_fetcher` (for a real use-case you would use `requests`), programmatically retrieve **ALL** of the reviews for a **single** restaurant (provided as a parameter). Just like the API was paginated, the HTML paginates its reviews (it would be a very long web page to show 300 reviews on a single page) and to get all the reviews you will need to parse and traverse the HTML. As input your function will receive a URL corresponding to a Yelp restaurant. As output return a list of dictionaries (structured the same as question 2.5) containing the relevant information from the reviews. You can use `parse_page()` here.

In [209]:
```
# 4% credits

def extract_reviews(url, html_fetcher):
    """
    Retrieve ALL of the reviews for a single restaurant on Yelp.

    Parameters:
        url (string): Yelp URL corresponding to the restaurant of interest.
        html_fetcher (function): A function that takes url and returns html

    Returns:
        reviews (list): list of dictionaries containing extracted review in
    """
    result = []
    while True:
        code, html = html_fetcher(url)
        reviews_list, url_next = parse_page(html)
        # replace url_next
        url = url_next
        # add that reviews to our arr
        result = result + reviews_list
        # if no more url, then stop
        if url == None:
            break

    return result
```

You can test your function with this code:

In [210]:
```
# 3% credits
data = extract_reviews('https://www.yelp.com/biz/the-jibarito-stop-chicago-
print(len(data))
# 35
print(data[0])
# {'author': 'Jason S.', 'rating': 5.0, 'date': '2016-05-02', 'description'
```

```
35
{'author': 'Jason S.', 'rating': 4.5, 'date': '2016-05-02', 'descriptio
n': "This was one of my favorite food trucks but as of last fall they've
opened a brick and mortar restaurant in the Pilsen neighborhood...the per
fect success story of how a person can start out with a food truck and gr
ow their business into a restaurant. The food is always delicious and the
service is great!\n"}
```

Submission

You're almost done!

After executing all commands and completing this notebook, save your *hw1.ipynb* as a pdf file and upload it to Gradescope under *Homework 1 (written)*. Make sure you check that your pdf file includes all parts of your solution **(including the outputs)**. We recommend using the browser (not jupyter) for saving the pdf. For Chrome on a Mac, this is under *File->Print...->Open PDF in Preview* and when the PDF opens in Preview you can use *Save...* to save it. This part will be graded based on completion (having executed the code and showing the output) and it constitutes *60%* of HW 1.

Next, you need to copy the functions from Questions 1.1 and 1.2 into the corresponding functions in *hw1part1.py*. Similarly, you need to copy the functions from Questions 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6 into the corresponding functions in *hw1part2.py*. Place your files *hw1part1.py*, *hw1part2.py*, and *hw1.ipynb* in a zip file and upload the zip file to Gradescope under *Homework 1 - (code)*. This part constitutes *40%* of HW 1. In order to get full points for this part, you need to pass all test cases that we will run against your *hw1part1.py* and *hw1part2.py* (and not the notebook) on Gradescope. We have provided a sample of the test cases in *tests_sample_part1/tests.py* and *tests_sample_part2/tests.py*. Other tests are hidden on the Gradescope server. To check whether your code runs locally, run the four tests in *tests_sample_part1* from your command line:

```
(cs418env) elena-macbook:hw1 elena$ python run_tests_sample.py part1
```

You should see the following output:

```
....
-------------------------------------------------------------------
---
Ran 4 tests in 0.001s

OK
```

Feel free to add more tests that check all parts of your code.

Similarly, you can run sample tests for part2 as follows:

```
(cs418env) elena-macbook:hw1 elena$ python run_tests_sample.py part2
```

You can submit to Gradescope as many times as you would like. We will only consider your last submission. If your last submission is after the deadline, the late homework policy applies.

After submitting the zip file, the autograder will run. You should see the following on your screen after the autograder finishes the execution:

This indicates that all the tests ran successfully on the server, and you're done! If your tests fail, you can debug your program locally by comparing the input, output and expected output (as shown for first two test cases). Make sure `hw1part1.py` , `hw1part2.py` and `hw1.ipynb` are included on the root of the zip file. **This means you need to zip those files and not the folder containing the files.**