pandas puzzles

short puzzles for testing your knowledge of pandas' power.

Since pandas is a large library with many different specialist features and functions, these excercises focus mainly on the fundamentals of manipulating data (indexing, grouping, aggregating, cleaning), making use of the core DataFrame and Series objects.

Many of the excerises here are stright-forward in that the solutions require no more than a few lines of code (in pandas or NumPy... don't go using pure Python or Cython!). Choosing the right methods and following best practices is the underlying goal.

The exercises are loosely divided in sections. Each section has a difficulty rating; these ratings are subjective, of course, but should be a seen as a rough guide as to how inventive the required solution is.

If you're just starting out with pandas and you are looking for some other resources, the official documentation is very extensive. In particular, some good places get a broader overview of pandas are...

- 10 minutes to pandas
- pandas basics
- tutorials
- cookbook and idioms

Enjoy the puzzles!

New Section

Importing pandas

Getting started and checking your pandas setup

Difficulty: *easy*

1. Import pandas under the alias pd.

```
import pandas as pd #1min
```

2. Print the version of pandas that has been imported.

```
print (pd.__version__) #3min
```

```
1.3.5
```

3. Print out all the *version* information of the libraries that are required by the pandas library.

```
pd.show_versions() #3min
```

```
INSTALLED VERSIONS
------------------
commit           : 66e3805b8cabe977f40c05259cc3fcf7ead5687d
python           : 3.7.13.final.0
python-bits      : 64
OS               : Linux
OS-release       : 5.4.188+
Version          : #1 SMP Sun Apr 24 10:03:06 PDT 2022
machine          : x86_64
processor        : x86_64
byteorder        : little
LC_ALL           : None
LANG             : en_US.UTF-8
LOCALE           : en_US.UTF-8

pandas           : 1.3.5
numpy            : 1.21.6
pytz             : 2022.1
dateutil         : 2.8.2
pip              : 21.1.3
setuptools       : 57.4.0
Cython           : 0.29.30
pytest           : 3.6.4
hypothesis       : None
sphinx           : 1.8.6
blosc            : None
feather          : 0.4.1
xlsxwriter       : None
lxml.etree       : 4.2.6
html5lib         : 1.0.1
pymysql          : None
psycopg2         : 2.7.6.1 (dt dec pq3 ext lo64)
jinja2           : 2.11.3
IPython          : 5.5.0
pandas_datareader: 0.9.0
bs4              : 4.6.3
bottleneck       : 1.3.4
fsspec           : None
fastparquet      : None
gcsfs            : None
matplotlib       : 3.2.2
numexpr          : 2.8.1
odfpy            : None
openpyxl         : 3.0.10
pandas_gbq       : 0.13.3
pyarrow          : 6.0.1
pyxlsb           : None
```

```
s3fs             : None
scipy            : 1.4.1
sqlalchemy       : 1.4.36
tables           : 3.7.0
tabulate         : 0.8.9
xarray           : 0.20.2
xlrd             : 1.1.0
xlwt             : 1.3.0
numba            : 0.51.2
```

DataFrame basics

A few of the fundamental routines for selecting, sorting, adding and aggregating data in DataFrames

Difficulty: *easy*

Note: remember to import numpy using:

```
import numpy as np
```

Consider the following Python dictionary `data` and Python list `labels`:

```
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat',
'snake', 'cat', 'dog', 'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no',
'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

(This is just some meaningless data I made up with the theme of animals and trips to a vet.)

4. Create a DataFrame `df` from this dictionary `data` which has the index `labels`.

```
import numpy as np

data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat',
'snake', 'cat', 'dog', 'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no',
'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(data, index=labels) #5min
```

5. Display a summary of the basic information about this DataFrame and its data (*hint: there is a single method that can be called on the DataFrame*).

```
df.info() # 3min
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, a to j
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   animal    10 non-null     object
 1   age       8 non-null      float64
 2   visits    10 non-null     int64
 3   priority  10 non-null     object
dtypes: float64(1), int64(1), object(2)
memory usage: 400.0+ bytes
```

6. Return the first 3 rows of the DataFrame `df`.

```
df.head(3) #1min
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
c  snake  0.5       2       no
```

7. Select just the 'animal' and 'age' columns from the DataFrame `df`.

```
df_sel_cols=df[['animal', 'age']]
print (df_sel_cols) #2min
```

```
  animal  age
a    cat  2.5
b    cat  3.0
c  snake  0.5
d    dog  NaN
e    dog  5.0
f    cat  2.0
g  snake  4.5
h    cat  NaN
i    dog  7.0
j    dog  3.0
```

8. Select the data in rows `[3, 4, 8]` *and* in columns `['animal', 'age']`.

```
df_sel_rows1 = df_sel_cols.iloc[[3, 4, 8]]
print (df_sel_rows1) #3min
```

```
  animal  age
d    dog  NaN
e    dog  5.0
i    dog  7.0
```

9. Select only the rows where the number of visits is greater than 3.

```
crit1=df['visits']>=3
df_sel_rows2 = df[crit1]
print (df_sel_rows2) #20min
```

```
  animal  age  visits priority
b    cat  3.0       3      yes
d    dog  NaN       3      yes
f    cat  2.0       3       no
```

10. Select the rows where the age is missing, i.e. it is `NaN`.

```
crit2 = df['age'].isnull()
df_sel_rows3 = df[crit2]
print (df_sel_rows3) #10min
```

```
  animal  age  visits priority
d    dog  NaN       3      yes
h    cat  NaN       1      yes
```

11. Select the rows where the animal is a cat *and* the age is less than 3.

```
crit4 = (df['animal']=='cat') & (df['age']<3)
df_sel_rows4 = df[crit4]
print (df_sel_rows4)  #5min
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
f    cat  2.0       3       no
```

12. Select the rows the age is between 2 and 4 (inclusive).

```
crit5 = (df['age'] >=2) & (df['age'] <=4)
df_sel_rows5 = df[crit5]
print (df_sel_rows5)   #5min
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
f    cat  2.0       3       no
j    dog  3.0       1       no
```

13. Change the age in row 'f' to 1.5.

```
df['age'].loc['f'] = 1.5
print (df) #10min
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
c  snake  0.5       2       no
d    dog  NaN       3      yes
e    dog  5.0       2       no
f    cat  1.5       3       no
```

```
g  snake  4.5       1       no
h    cat  NaN       1      yes
i    dog  7.0       2       no
j    dog  3.0       1       no
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/indexing.py:1732:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

14. Calculate the sum of all visits in `df` (i.e. find the total number of visits).

```
sum_visits = df['visits'].sum()
print (sum_visits) #2min
```

```
19
```

15. Calculate the mean age for each different animal in `df`.

```
df_grouped=df.groupby(['animal'])['age'].agg('mean')
print (df_grouped) #10min
```

```
animal
cat      2.333333
dog      5.000000
snake    2.500000
Name: age, dtype: float64
```

16. Append a new row 'k' to `df` with your choice of values for each column. Then delete that row to return the original DataFrame.

```
new_row = ['goat', 1.5, 2, 'yes']
df.loc['k'] = new_row
#print(df)
df=df.drop(['k'])
print(df) #20min
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
c  snake  0.5       2       no
d    dog  NaN       3      yes
e    dog  5.0       2       no
f    cat  1.5       3       no
g  snake  4.5       1       no
h    cat  NaN       1      yes
i    dog  7.0       2       no
j    dog  3.0       1       no
```

17. Count the number of each type of animal in `df`.

```
df_grouped1 = df.groupby(['animal'])['animal'].agg('count')
print (df_grouped1) #5min
```

```
animal
cat      4
dog      4
snake    2
Name: animal, dtype: int64
```

18. Sort `df` first by the values in the 'age' in *decending* order, then by the value in the 'visits' column in *ascending* order (so row `i` should be first, and row `d` should be last).

```
df_sorted = df.sort_values(['age', 'visits'], ascending=[False, True])
print (df_sorted) #5min
```

```
  animal  age  visits priority
i    dog  7.0       2       no
e    dog  5.0       2       no
g  snake  4.5       1       no
j    dog  3.0       1       no
b    cat  3.0       3      yes
a    cat  2.5       1      yes
f    cat  1.5       3       no
c  snake  0.5       2       no
h    cat  NaN       1      yes
d    dog  NaN       3      yes
```

19. The 'priority' column contains the values 'yes' and 'no'. Replace this column with a column of boolean values: 'yes' should be `True` and 'no' should be `False`.

```
df['priority'] = df['priority'].replace(['yes', 'no'], ['True',
'False'])
print (df) #20min
```

```
  animal  age  visits priority
a    cat  2.5       1     True
b    cat  3.0       3     True
c  snake  0.5       2    False
d    dog  NaN       3     True
e    dog  5.0       2    False
f    cat  1.5       3    False
g  snake  4.5       1    False
h    cat  NaN       1     True
i    dog  7.0       2    False
j    dog  3.0       1    False
```

20. In the 'animal' column, change the 'snake' entries to 'python'.

```
df['animal'] = df['animal'].replace('snake', 'python')
print(df) #10min
```

```
   animal  age  visits priority
a     cat  2.5       1     True
b     cat  3.0       3     True
c  python  0.5       2    False
d     dog  NaN       3     True
e     dog  5.0       2    False
f     cat  1.5       3    False
g  python  4.5       1    False
h     cat  NaN       1     True
i     dog  7.0       2    False
j     dog  3.0       1    False
```

21. For each animal type and each number of visits, find the mean age. In other words, each row is an animal, each column is a number of visits and the values are the mean ages (*hint: use a pivot table*).

```
df_piv=pd.pivot_table(df, index=df['animal'])
print(df_piv) #10min
```

```
             age  visits
animal
cat     2.333333     2.0
dog     5.000000     2.0
python  2.500000     1.5
```

DataFrames: beyond the basics

Slightly trickier: you may need to combine two or more methods to get the right answer

Difficulty: *medium*

The previous section was tour through some basic but essential DataFrame operations. Below are some ways that you might need to cut your data, but for which there is no single "out of the box" method.

22. You have a DataFrame `df` with a column 'A' of integers. For example:

```
df = pd.DataFrame({'A': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7, 7]})
```

How do you filter out rows which contain the same integer as the row immediately above?

You should be left with a column containing the following values:

```
1, 2, 3, 4, 5, 6, 7
```

```
df = pd.DataFrame({'A': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7, 7]})
df_filt = df.loc[df['A'].shift() != df['A']]
print(df_filt) #20min
```

```
   A
0  1
1  2
```

```
3  3
4  4
5  5
8  6
9  7
```

23. Given a DataFrame of numeric values, say

```
df = pd.DataFrame(np.random.random(size=(5, 3))) # a 5x3 frame of
float values
```

how do you subtract the row mean from each element in the row?

```
df = pd.DataFrame(np.random.random(size=(5, 3)))
df_sub = df.sub(df.mean(axis=1), axis=0)
print (df_sub) #20min
```

```
          0         1         2
0 -0.025186 -0.000606  0.025792
1  0.399801 -0.043536 -0.356265
2  0.204901 -0.020128 -0.184774
3 -0.204229  0.192870  0.011359
4 -0.569243  0.339169  0.230074
```

24. Suppose you have DataFrame with 10 columns of real numbers, for example:

```
df = pd.DataFrame(np.random.random(size=(5, 10)),
columns=list('abcdefghij'))
```

Which column of numbers has the smallest sum? Return that column's label.

```
df = pd.DataFrame(np.random.random(size=(5, 10)),
columns=list('abcdefghij'))
col_label = df.sum().idxmin()
print (col_label) #2min
```

```
d
```

25. How do you count how many unique rows a DataFrame has (i.e. ignore all rows that are duplicates)? As input, use a DataFrame of zeros and ones with 10 rows and 3 columns.

```
df = pd.DataFrame(np.random.randint(0, 2, size=(10, 3)))
```

```
df = pd.DataFrame(np.random.randint(0, 2, size=(10, 3)))
unique_rows = len(df) - df.duplicated(keep = False).sum()
print(unique_rows) #10min
```

```
4
```

The next three puzzles are slightly harder.

26. In the cell below, you have a DataFrame `df` that consists of 10 columns of floating-point numbers. Exactly 5 entries in each row are NaN values.

For each row of the DataFrame, find the *column* which contains the *third* NaN value.

You should return a Series of column labels: `e, c, d, h, d`

```
nan = np.nan

data = [[0.04,  nan,  nan, 0.25,  nan, 0.43, 0.71, 0.51,  nan,  nan],
        [ nan,  nan,  nan, 0.04, 0.76,  nan,  nan, 0.67, 0.76, 0.16],
        [ nan,  nan, 0.5 ,  nan, 0.31, 0.4 ,  nan,  nan, 0.24, 0.01],
        [0.49,  nan,  nan, 0.62, 0.73, 0.26, 0.85,  nan,  nan,  nan],
        [ nan,  nan, 0.41,  nan, 0.05,  nan, 0.61,  nan, 0.48, 0.68]]

columns = list('abcdefghij')

df = pd.DataFrame(data, columns=columns)

# write a solution to the question here
df_thirdNaN=(df.isnull().cumsum(axis=1) == 3).idxmax(axis=1)
print (df_thirdNaN) #15min

0    e
1    c
2    d
3    h
4    d
dtype: object
```

27. A DataFrame has a column of groups 'grps' and and column of integer values 'vals':

```
df = pd.DataFrame({'grps': list('aaabbcaabcccbbc'),
                   'vals':
[12,345,3,1,45,14,4,52,54,23,235,21,57,3,87]})
```

For each *group*, find the sum of the three greatest values. You should end up with the answer as follows:

```
grps
a    409
b    156
c    345

df = pd.DataFrame({'grps': list('aaabbcaabcccbbc'),
                   'vals':
[12,345,3,1,45,14,4,52,54,23,235,21,57,3,87]})

# write a solution to the question here
df=df.groupby('grps')['vals'].nlargest(3).sum(level=0)
print(df) #15min

grps
a    409
```

```
b    156
c    345
Name: vals, dtype: int64

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5:
FutureWarning: Using the level keyword in DataFrame and Series
aggregations is deprecated and will be removed in a future version.
Use groupby instead. df.sum(level=1) should use
df.groupby(level=1).sum().
  """
```

28. The DataFrame `df` constructed below has two integer columns 'A' and 'B'. The values in 'A' are between 1 and 100 (inclusive).

For each group of 10 consecutive integers in 'A' (i.e. `(0, 10]`, `(10, 20]`, ...), calculate the sum of the corresponding values in column 'B'.

The answer should be a Series as follows:

```
A
(0, 10]      635
(10, 20]     360
(20, 30]     315
(30, 40]     306
(40, 50]     750
(50, 60]     284
(60, 70]     424
(70, 80]     526
(80, 90]     835
(90, 100]    852
```

```
df = pd.DataFrame(np.random.RandomState(8765).randint(1, 101,
size=(100, 2)), columns = ["A", "B"])

# write a solution to the question here
df = df.groupby(pd.cut(df['A'], np.arange(0, 101, 10)))['B'].sum()
print(df) #20min
```

```
A
(0, 10]      635
(10, 20]     360
(20, 30]     315
(30, 40]     306
(40, 50]     750
(50, 60]     284
(60, 70]     424
(70, 80]     526
(80, 90]     835
(90, 100]    852
Name: B, dtype: int64
```

DataFrames: harder problems

These might require a bit of thinking outside the box...

...but all are solvable using just the usual pandas/NumPy methods (and so avoid using explicit `for` loops).

Difficulty: *hard*

29. Consider a DataFrame `df` where there is an integer column 'X':

```
df = pd.DataFrame({'X': [7, 2, 0, 3, 4, 2, 5, 0, 3, 4]})
```

For each value, count the difference back to the previous zero (or the start of the Series, whichever is closer). These values should therefore be

```
[1, 2, 0, 1, 2, 3, 4, 0, 1, 2]
```

Make this a new column 'Y'.

```
df = pd.DataFrame({'X': [7, 2, 0, 3, 4, 2, 5, 0, 3, 4]})

x = (df['X'] != 0).cumsum()
y = x != x.shift()
df['Y'] = y.groupby((y != y.shift()).cumsum()).cumsum()

print(df) #30min
```

```
   X  Y
0  7  1
1  2  2
2  0  0
3  3  1
4  4  2
5  2  3
6  5  4
7  0  0
8  3  1
9  4  2
```

30. Consider the DataFrame constructed below which contains rows and columns of numerical data.

Create a list of the column-row index locations of the 3 largest values in this DataFrame. In this case, the answer should be:

```
[(5, 7), (6, 4), (2, 5)]
```

31. You are given the DataFrame below with a column of group IDs, 'grps', and a column of corresponding integer values, 'vals'.

```
df = pd.DataFrame({"vals": np.random.RandomState(31).randint(-30, 30,
size=15),
                   "grps": np.random.RandomState(31).choice(["A",
"B"], 15)})
```

Create a new column 'patched_values' which contains the same values as the 'vals' any negative values in 'vals' with the group mean:

```
    vals grps  patched_vals
0    -12    A          13.6
1     -7    B          28.0
2    -14    A          13.6
3      4    A           4.0
4     -7    A          13.6
5     28    B          28.0
6     -2    A          13.6
7     -1    A          13.6
8      8    A           8.0
9     -2    B          28.0
10    28    A          28.0
11    12    A          12.0
12    16    A          16.0
13   -24    A          13.6
14   -12    A          13.6
```

32. Implement a rolling mean over groups with window size 3, which ignores NaN value. For example consider the following DataFrame:

```
>>> df = pd.DataFrame({'group': list('aabbabbbabab'),
                       'value': [1, 2, 3, np.nan, 2, 3, np.nan, 1, 7,
3, np.nan, 8]})
>>> df
   group  value
0      a    1.0
1      a    2.0
2      b    3.0
3      b    NaN
4      a    2.0
5      b    3.0
6      b    NaN
7      b    1.0
8      a    7.0
9      b    3.0
10     a    NaN
11     b    8.0
```

The goal is to compute the Series:

```
0     1.000000
1     1.500000
2     3.000000
3     3.000000
4     1.666667
5     3.000000
6     3.000000
7     2.000000
8     3.666667
9     2.000000
10    4.500000
11    4.000000
```

E.g. the first window of size three for group 'b' has values 3.0, NaN and 3.0 and occurs at row index 5. Instead of being NaN the value in the new column at this row index should be 3.0 (just the two non-NaN values are used to compute the mean (3+3)/2)

Series and DatetimeIndex

Exercises for creating and manipulating Series with datetime data

Difficulty: *easy/medium*

pandas is fantastic for working with dates and times. These puzzles explore some of this functionality.

33. Create a DatetimeIndex that contains each business day of 2015 and use it to index a Series of random numbers. Let's call this Series `s`.

```
dti = pd.date_range(start='2015-01-01', end='2015-12-31', freq='B')
s = pd.Series(np.random.rand(len(dti)), index=dti)
print(s) #15min
```

```
2015-01-01    0.409074
2015-01-02    0.978121
2015-01-05    0.566613
2015-01-06    0.058111
2015-01-07    0.512211
                ...
2015-12-25    0.576461
2015-12-28    0.224682
2015-12-29    0.430470
2015-12-30    0.946864
2015-12-31    0.457968
Freq: B, Length: 261, dtype: float64
```

34. Find the sum of the values in `s` for every Wednesday.

```
sum = s[s.index.weekday == 2].sum()
print(sum) #5min
```

```
26.695687888737183
```

35. For each calendar month in `s`, find the mean of values.

```
mean_vals = s.resample('M').mean()
print(mean_vals) #5min
```

```
2015-01-31    0.559769
2015-02-28    0.472341
2015-03-31    0.538845
2015-04-30    0.506323
2015-05-31    0.489136
2015-06-30    0.414893
2015-07-31    0.473189
2015-08-31    0.511495
2015-09-30    0.507998
2015-10-31    0.562078
2015-11-30    0.531517
2015-12-31    0.547604
Freq: M, dtype: float64
```

36. For each group of four consecutive calendar months in `s`, find the date on which the highest value occurred.

```
mean_val = s.resample('M').mean()
print(mean_val) #5min
```

```
2015-01-31    0.559769
2015-02-28    0.472341
2015-03-31    0.538845
2015-04-30    0.506323
2015-05-31    0.489136
2015-06-30    0.414893
2015-07-31    0.473189
2015-08-31    0.511495
2015-09-30    0.507998
2015-10-31    0.562078
2015-11-30    0.531517
2015-12-31    0.547604
Freq: M, dtype: float64
```

37. Create a DateTimeIndex consisting of the third Thursday in each month for the years 2015 and 2016.

```
idx = pd.date_range('2015-01-01', '2016-12-31', freq='WOM-3THU')
print(idx) #10min
```

```
DatetimeIndex(['2015-01-15', '2015-02-19', '2015-03-19', '2015-04-16',
               '2015-05-21', '2015-06-18', '2015-07-16', '2015-08-20',
               '2015-09-17', '2015-10-15', '2015-11-19', '2015-12-17',
               '2016-01-21', '2016-02-18', '2016-03-17', '2016-04-21',
               '2016-05-19', '2016-06-16', '2016-07-21', '2016-08-18',
```

```
             '2016-09-15', '2016-10-20', '2016-11-17', '2016-12-
15'],
              dtype='datetime64[ns]', freq='WOM-3THU')
```

Cleaning Data

Making a DataFrame easier to work with

Difficulty: *easy/medium*

It happens all the time: someone gives you data containing malformed strings, Python, lists and missing data. How do you tidy it up so you can get on with the analysis?

Take this monstrosity as the DataFrame to use in the following puzzles:

```
df = pd.DataFrame({'From_To': ['LoNDon_paris', 'MAdrid_miLAN',
'londON_StockhOlm',
                               'Budapest_PaRis', 'Brussels_londOn'],
              'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],
              'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67,
32]],
                   'Airline': ['KLM(!)', '<Air France> (12)',
'(British Airways. )',
                               '12. Air France', '"Swiss Air"']})
```

Formatted, it looks like this:

```
            From_To  FlightNumber  RecentDelays              Airline
0      LoNDon_paris       10045.0      [23, 47]               KLM(!)
1      MAdrid_miLAN           NaN            []    <Air France> (12)
2  londON_StockhOlm       10065.0  [24, 43, 87]  (British Airways. )
3    Budapest_PaRis           NaN          [13]       12. Air France
4   Brussels_londOn       10085.0      [67, 32]          "Swiss Air"
```

(It's some flight data I made up; it's not meant to be accurate in any way.)

38. Some values in the the **FlightNumber** column are missing (they are `NaN`). These numbers are meant to increase by 10 with each row so 10055 and 10075 need to be put in place. Modify `df` to fill in these missing numbers and make the column an integer column (instead of a float column).

```
df = pd.DataFrame({'From_To': ['LoNDon_paris', 'MAdrid_miLAN',
'londON_StockhOlm',
                               'Budapest_PaRis', 'Brussels_londOn'],
              'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],
              'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67,
32]],
                   'Airline': ['KLM(!)', '<Air France> (12)',
'(British Airways. )',
                               '12. Air France', '"Swiss Air"']})
```

```
df['FlightNumber'] = df['FlightNumber'].interpolate().astype(int)
print(df['FlightNumber'])  #15min
```

```
0    10045
1    10055
2    10065
3    10075
4    10085
Name: FlightNumber, dtype: int64
```

39. The **From_To** column would be better as two separate columns! Split each string on the underscore delimiter _ to give a new temporary DataFrame called 'temp' with the correct values. Assign the correct column names 'From' and 'To' to this temporary DataFrame.

```
temp = df.From_To.str.split('_', expand=True)
temp.columns = ['From', 'To']
print(temp)   #5min
```

```
       From         To
0    LoNDon      paris
1    MAdrid      miLAN
2    londON  StockhOlm
3  Budapest      PaRis
4  Brussels     londOn
```

40. Notice how the capitalisation of the city names is all mixed up in this temporary DataFrame 'temp'. Standardise the strings so that only the first letter is uppercase (e.g. "londON" should become "London".)

```
temp['From'] = temp['From'].str.capitalize()
temp['To'] = temp['To'].str.capitalize()
print(temp)   #5min
```

```
       From         To
0    London      Paris
1    Madrid      Milan
2    London  Stockholm
3  Budapest      Paris
4  Brussels     London
```

41. Delete the **From_To** column from `df` and attach the temporary DataFrame 'temp' from the previous questions.

```
df = df.drop('From_To', axis=1)
df = df.join(temp)
print(df)    #5min
```

```
   FlightNumber  RecentDelays              Airline      From
To
0         10045      [23, 47]               KLM(!)    London
Paris
1         10055            []    <Air France> (12)    Madrid
```

```
Milan
2         10065  [24, 43, 87]  (British Airways. )    London
Stockholm
3         10075          [13]       12. Air France  Budapest
Paris
4         10085      [67, 32]          "Swiss Air"  Brussels
London
```

42. In the **Airline** column, you can see some extra puctuation and symbols have appeared around the airline names. Pull out just the airline name. E.g. `'(British Airways. )'` should become `'British Airways'`.

```
df['Airline'] = df['Airline'].str.extract('([a-zA-Z\s]+)',
expand=False).str.strip()
print(df['Airline'])  #20min
```

```
0                 KLM
1          Air France
2     British Airways
3          Air France
4           Swiss Air
Name: Airline, dtype: object
```

43. In the RecentDelays column, the values have been entered into the DataFrame as a list. We would like each first value in its own column, each second value in its own column, and so on. If there isn't an Nth value, the value should be NaN.

Expand the Series of lists into a DataFrame named `delays`, rename the columns `delay_1`, `delay_2`, etc. and replace the unwanted RecentDelays column in `df` with `delays`.

```
delays = df['RecentDelays'].apply(pd.Series)
delays.columns = ['delay_{}'.format(n) for n in range(1,
len(delays.columns)+1)]
df = df.drop('RecentDelays', axis=1).join(delays)
print(df)  #30min
```

```
   FlightNumber          Airline      From         To  delay_1
delay_2  \
0         10045              KLM    London      Paris     23.0
47.0
1         10055       Air France    Madrid      Milan      NaN
NaN
2         10065  British Airways    London  Stockholm     24.0
43.0
3         10075       Air France  Budapest      Paris     13.0
NaN
4         10085        Swiss Air  Brussels     London     67.0
32.0

   delay_3
0      NaN
```

```
1      NaN
2     87.0
3      NaN
4      NaN
```

The DataFrame should look much better now.

```
   FlightNumber          Airline      From         To  delay_1
delay_2  delay_3
0         10045              KLM    London      Paris     23.0
47.0      NaN
1         10055       Air France    Madrid      Milan      NaN
NaN      NaN
2         10065  British Airways    London  Stockholm     24.0
43.0     87.0
3         10075       Air France  Budapest      Paris     13.0
NaN      NaN
4         10085        Swiss Air  Brussels     London     67.0
32.0      NaN
```

Using MultiIndexes

Go beyond flat DataFrames with additional index levels

Difficulty: *medium*

Previous exercises have seen us analysing data from DataFrames equipped with a single index level. However, pandas also gives you the possibilty of indexing your data using *multiple* levels. This is very much like adding new dimensions to a Series or a DataFrame. For example, a Series is 1D, but by using a MultiIndex with 2 levels we gain of much the same functionality as a 2D DataFrame.

The set of puzzles below explores how you might use multiple index levels to enhance data analysis.

To warm up, we'll look make a Series with two index levels.

44. Given the lists `letters = ['A', 'B', 'C']` and `numbers = list(range(10))`, construct a MultiIndex object from the product of the two lists. Use it to index a Series of random numbers. Call this Series `s`.

```
import pandas as pd

letters = ['A', 'B', 'C']
numbers = list(range(10))

multi_idx = pd.MultiIndex.from_product([letters, numbers])
s = pd.Series(np.random.rand(30), index=multi_idx)
print(s) #20min
```

```
A  0    0.140618
   1    0.687915
   2    0.111853
   3    0.636129
   4    0.588547
   5    0.189700
   6    0.888148
   7    0.068398
   8    0.056247
   9    0.771034
B  0    0.067540
   1    0.932084
   2    0.619941
   3    0.803332
   4    0.610752
   5    0.276955
   6    0.467640
   7    0.148595
   8    0.813688
   9    0.967769
C  0    0.205735
   1    0.726326
   2    0.230221
   3    0.055780
   4    0.440057
   5    0.626687
   6    0.230927
   7    0.747400
   8    0.092317
   9    0.120675
dtype: float64
```

45. Check the index of `s` is lexicographically sorted (this is a necessary proprty for indexing to work correctly with a MultiIndex).

```
statement = s.index.is_lexsorted()
print(statement)
```

```
True
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1:
FutureWarning: MultiIndex.is_lexsorted is deprecated as a public
function, users should use MultiIndex.is_monotonic_increasing instead.
  """Entry point for launching an IPython kernel.
```

46. Select the labels `1`, `3` and `6` from the second level of the MultiIndexed Series.

```
sel_indices = s.loc[:, [1, 3, 6]]
print (sel_indices) #10min
```

```
A  1    0.687915
   3    0.636129
```

```
   6    0.888148
B  1    0.932084
   3    0.803332
   6    0.467640
C  1    0.726326
   3    0.055780
   6    0.230927
dtype: float64
```

47. Slice the Series `s`; slice up to label 'B' for the first level and from label 5 onwards for the second level.

```
sliced_indices = s.loc[pd.IndexSlice[:'B', 5:]]
print (sliced_indices)  #15min
```

```
A  5    0.189700
   6    0.888148
   7    0.068398
   8    0.056247
   9    0.771034
B  5    0.276955
   6    0.467640
   7    0.148595
   8    0.813688
   9    0.967769
dtype: float64
```

48. Sum the values in `s` for each label in the first level (you should have Series giving you a total for labels A, B and C).

```
sum_indices_lev0 = s.sum(level=0)
print(sum_indices_lev0)  #15min
```

```
A    4.138590
B    5.708295
C    3.476123
dtype: float64
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1:
FutureWarning: Using the level keyword in DataFrame and Series
aggregations is deprecated and will be removed in a future version.
Use groupby instead. df.sum(level=1) should use
df.groupby(level=1).sum().
  """Entry point for launching an IPython kernel.
```

49. Suppose that `sum()` (and other methods) did not accept a `level` keyword argument. How else could you perform the equivalent of `s.sum(level=1)`?

```
sum_indices_lev1 = s.unstack().sum(axis=0)
print(sum_indices_lev1) #15min
```

```
0    0.413893
1    2.346325
2    0.962015
3    1.495240
4    1.639356
5    1.093342
6    1.586714
7    0.964393
8    0.962252
9    1.859477
dtype: float64
```

50. Exchange the levels of the MultiIndex so we have an index of the form (letters, numbers). Is this new Series properly lexsorted? If not, sort it.

```
new_s = s.swaplevel(0, 1)
#is lexsorted?
is_lexsorted = new_s.index.is_lexsorted()
print(is_lexsorted)
#sort it
new_s = new_s.sort_index()

print(new_s) #15min
```

```
False
0  A    0.140618
   B    0.067540
   C    0.205735
1  A    0.687915
   B    0.932084
   C    0.726326
2  A    0.111853
   B    0.619941
   C    0.230221
3  A    0.636129
   B    0.803332
   C    0.055780
4  A    0.588547
   B    0.610752
   C    0.440057
5  A    0.189700
   B    0.276955
   C    0.626687
6  A    0.888148
   B    0.467640
   C    0.230927
7  A    0.068398
   B    0.148595
   C    0.747400
8  A    0.056247
   B    0.813688
```

```
   C    0.092317
9  A    0.771034
   B    0.967769
   C    0.120675
dtype: float64
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
FutureWarning: MultiIndex.is_lexsorted is deprecated as a public
function, users should use MultiIndex.is_monotonic_increasing instead.
  This is separate from the ipykernel package so we can avoid doing
imports until
```

Minesweeper

Generate the numbers for safe squares in a Minesweeper grid

Difficulty: *medium* to *hard*

If you've ever used an older version of Windows, there's a good chance you've played with Minesweeper:

- https://en.wikipedia.org/wiki/Minesweeper_(video_game)

If you're not familiar with the game, imagine a grid of squares: some of these squares conceal a mine. If you click on a mine, you lose instantly. If you click on a safe square, you reveal a number telling you how many mines are found in the squares that are immediately adjacent. The aim of the game is to uncover all squares in the grid that do not contain a mine.

In this section, we'll make a DataFrame that contains the necessary data for a game of Minesweeper: coordinates of the squares, whether the square contains a mine and the number of mines found on adjacent squares.

51. Let's suppose we're playing Minesweeper on a 5 by 4 grid, i.e.

```
X = 5
Y = 4
```

To begin, generate a DataFrame `df` with two columns, `'x'` and `'y'` containing every coordinate for this grid. That is, the DataFrame should start:

```
   x  y
0  0  0
1  0  1
2  0  2
```

```
import pandas as pd
# Dataframe as a response to the question above df.shape() = (3, 2)
df = pd.DataFrame({'x': [0, 0, 0], 'y': [0, 1, 2]})
print (df.shape)
#Actual Dataframe is a grid with X=5, Y=4, meaning (x = [0, 1, 2, 3,
4]), (y = [0, 1, 2, 3]) which leads to a list of tuples  (x, y) like
```

```
this:

coord_lst = [(0,0), (1, 0), (2, 0), (3, 0), (4, 0), (0, 1), (1, 1),
(2, 1), (3, 1), (4, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (0,
3), (1, 3), (2, 3), (3, 3), (4, 3)]
X = [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
Y = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3]
df = pd.DataFrame({'X': X,'Y':Y}) #df.size = 5*4 = 20
print (df) #30min
```

```
(3, 2)
    X  Y
0   0  0
1   1  0
2   2  0
3   3  0
4   4  0
5   0  1
6   1  1
7   2  1
8   3  1
9   4  1
10  0  2
11  1  2
12  2  2
13  3  2
14  4  2
15  0  3
16  1  3
17  2  3
18  3  3
19  4  3
```

52. For this DataFrame `df`, create a new column of zeros (safe) and ones (mine). The probability of a mine occuring at each location should be 0.4.

```
#initially all points are safe (we don't know where mines are) and
prob_mine = 0.4 for all safe points. Additionally, we create a list
'mine' were 1 means there is a mine, 0 = no mine
df_state_0 = df.assign(safe = [0]*20, mine = [0, 0, 1, 0, 1, 0, 0, 0,
1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1], prob_mine = [0.4]*20) #initial
state of the grid
print(df_state_0) #30min
```

```
    X  Y  safe  mine  prob_mine
0   0  0     0     0        0.4
1   1  0     0     0        0.4
2   2  0     0     1        0.4
3   3  0     0     0        0.4
4   4  0     0     1        0.4
5   0  1     0     0        0.4
```

```
6   1  1     0     0        0.4
7   2  1     0     0        0.4
8   3  1     0     1        0.4
9   4  1     0     1        0.4
10  0  2     0     0        0.4
11  1  2     0     0        0.4
12  2  2     0     1        0.4
13  3  2     0     0        0.4
14  4  2     0     1        0.4
15  0  3     0     0        0.4
16  1  3     0     0        0.4
17  2  3     0     1        0.4
18  3  3     0     0        0.4
19  4  3     0     1        0.4
```

53. Now create a new column for this DataFrame called `'adjacent'`. This column should contain the number of mines found on adjacent squares in the grid.

(E.g. for the first row, which is the entry for the coordinate `(0, 0)`, count how many mines are found on the coordinates `(0, 1)`, `(1, 0)` and `(1, 1)`.)

```
#Initial game model
#column 'adjacent' is dynamic, it holds the transformation of the grid
as we progressivelly reveal more mines(mine =1 for the  respective
coordinate,
#safe=1(no longer safe) and prob_mine =1-definitely there is a mine
there) or when we enter the final catastrophic state
#when we step on a mine, case in which entire column 'safe' becomes 1,
mine' becomes 1, 'prob_mine' becomes 1 and for which game is over with
a LOSS
#(final state with LOSS, FSL)
#Game also enters the final state with WIN (FSW) when we manage to
reveal all dangerous places in the grid, using the mechanisms of the
game and the succesive
#changes/transitions described above, without entering the FSL, that
is transitioning succesfully to either one of the following states
described below:
#MP(safe =>(0->1), mine =>(0->1), prob_mine => (0.4->1)) (mine
presence MP) or MA(safe => (0->0), mine => (0->0), prob_mine => (0.4-
>0)) (mine absence MA)

#So FSW is our DataFrame with rows of types (0, 0, 0) or (1, 1, 1) for
the columns [safe, mine, prob_mine] and
#FWL is our DataFrame with all rows of types (1, 1, 1) for the columns
[safe, mine, prob_mine]

#Counting the mines in each row on 'adjacent' column, means to
calculate sum() over all the adjacent coordinates for which we have a
modified state
#Marginal coordinates (of type (0, y) or (x, 0), a formation of _|_)
```

```
have only 3 neighbours for which we need to calculate the sum, whereas

#inner coordinates have 4 neighbours of type (x, y) with x != 0, y !
=0, making a cross like formation, have 4 neighbours for which we need
to calculate the sum
# We create 2 criteria for filtering the pairs of coordinates of df,
accordingly:
# Initially, we'll populate this column with zeros.
df_state_0 = df_state_0.assign(adjacent = [0]*20)
print(df_state_0) #120min
```

```
    X  Y  safe  mine  prob_mine  adjacent
0   0  0     0     0        0.4         0
1   1  0     0     0        0.4         0
2   2  0     0     1        0.4         0
3   3  0     0     0        0.4         0
4   4  0     0     1        0.4         0
5   0  1     0     0        0.4         0
6   1  1     0     0        0.4         0
7   2  1     0     0        0.4         0
8   3  1     0     1        0.4         0
9   4  1     0     1        0.4         0
10  0  2     0     0        0.4         0
11  1  2     0     0        0.4         0
12  2  2     0     1        0.4         0
13  3  2     0     0        0.4         0
14  4  2     0     1        0.4         0
15  0  3     0     0        0.4         0
16  1  3     0     0        0.4         0
17  2  3     0     1        0.4         0
18  3  3     0     0        0.4         0
19  4  3     0     1        0.4         0
```

54. For rows of the DataFrame that contain a mine, set the value in the `'adjacent'` column to NaN.

```
df_state_0.loc[df_state_0['mine']==1, 'adjacent'] = np.nan

print(df_state_0) #30min
```

```
    X  Y  safe  mine  prob_mine  adjacent
0   0  0     0     0        0.4       0.0
1   1  0     0     0        0.4       0.0
2   2  0     0     1        0.4       NaN
3   3  0     0     0        0.4       0.0
4   4  0     0     1        0.4       NaN
5   0  1     0     0        0.4       0.0
6   1  1     0     0        0.4       0.0
7   2  1     0     0        0.4       0.0
8   3  1     0     1        0.4       NaN
9   4  1     0     1        0.4       NaN
```

```
10  0  2     0     0        0.4       0.0
11  1  2     0     0        0.4       0.0
12  2  2     0     1        0.4       NaN
13  3  2     0     0        0.4       0.0
14  4  2     0     1        0.4       NaN
15  0  3     0     0        0.4       0.0
16  1  3     0     0        0.4       0.0
17  2  3     0     1        0.4       NaN
18  3  3     0     0        0.4       0.0
19  4  3     0     1        0.4       NaN
```

55. Finally, convert the DataFrame to grid of the adjacent mine counts: columns are the x coordinate, rows are the y coordinate.

```
df_state_0 = df_state_0.set_index(['Y', 'X']).unstack()
print(df_state_0) #20min
```

```
  safe                mine                prob_mine
adjacent  \
X    0  1  2  3  4    0  1  2  3  4          0    1    2    3    4
0
Y

0    0  0  0  0  0    0  0  1  0  1       0.4  0.4  0.4  0.4  0.4
0.0
1    0  0  0  0  0    0  0  0  1  1       0.4  0.4  0.4  0.4  0.4
0.0
2    0  0  0  0  0    0  0  1  0  1       0.4  0.4  0.4  0.4  0.4
0.0
3    0  0  0  0  0    0  0  1  0  1       0.4  0.4  0.4  0.4  0.4
0.0


X    1    2    3   4
Y
0  0.0  NaN  0.0 NaN
1  0.0  0.0  NaN NaN
2  0.0  NaN  0.0 NaN
3  0.0  NaN  0.0 NaN
```

Plotting

Visualize trends and patterns in data

Difficulty: *medium*

To really get a good understanding of the data contained in your DataFrame, it is often essential to create plots: if you're lucky, trends and anomalies will jump right out at you. This functionality is baked into pandas and the puzzles below explore some of what's possible with the library.

56. Pandas is highly integrated with the plotting library matplotlib, and makes plotting DataFrames very user-friendly! Plotting in a notebook environment usually makes use of the following boilerplate:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
```

matplotlib is the plotting library which pandas' plotting functionality is built upon, and it is usually aliased to `plt`.

`%matplotlib inline` tells the notebook to show plots inline, instead of creating them in a separate window.

`plt.style.use('ggplot')` is a style theme that most people find agreeable, based upon the styling of R's ggplot package.

For starters, make a scatter plot of this random data, but use black X's instead of the default markers.

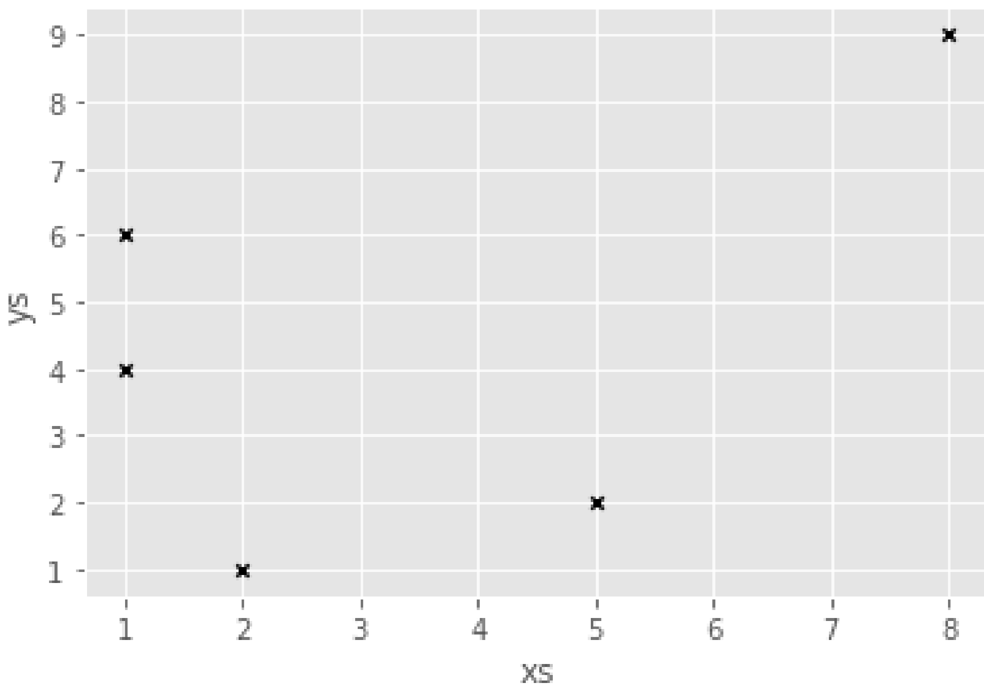
`df = pd.DataFrame({"xs":[1,5,2,8,1], "ys":[4,2,1,9,6]})`

Consult the documentation if you get stuck!

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')

df = pd.DataFrame({'xs':[1, 5, 2, 8, 1], 'ys':[4, 2, 1, 9, 6]})
df.plot.scatter('xs', "ys", color = 'black', marker = "x" )  #10min
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb97b04c790>
```



57. Columns in your DataFrame can also be used to modify colors and sizes. Bill has been keeping track of his performance at work over time, as well as how good he was feeling that day, and whether he had a cup of coffee in the morning. Make a plot which incorporates all four features of this DataFrame.

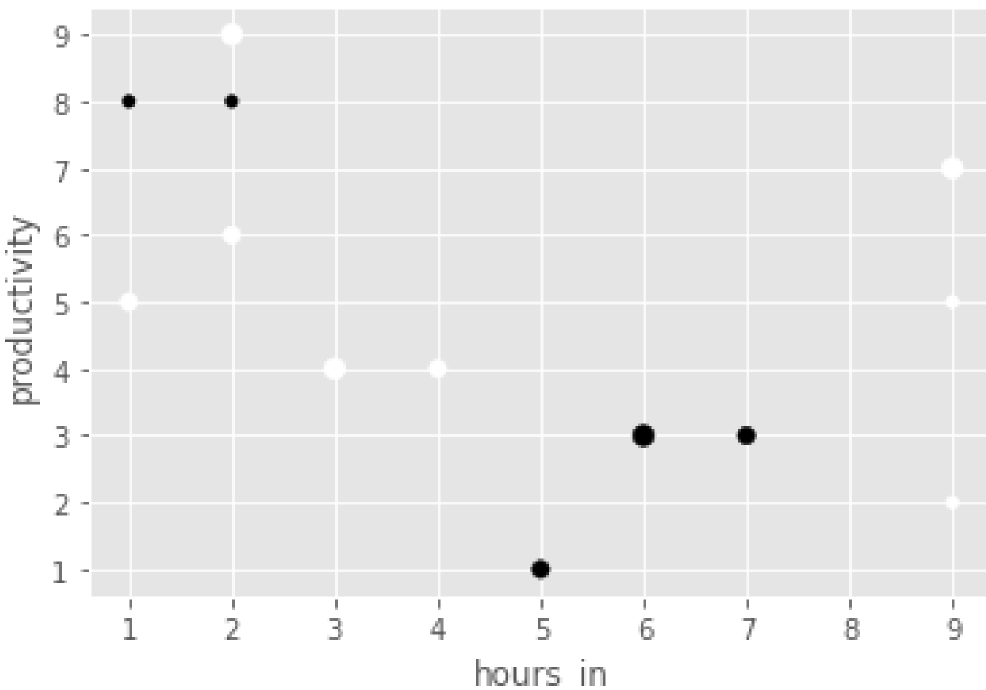
(Hint: If you're having trouble seeing the plot, try multiplying the Series which you choose to represent size by 10 or more)

The chart doesn't have to be pretty: this isn't a course in data viz!

```
df = pd.DataFrame({"productivity":[5,2,3,1,4,5,6,7,8,3,4,8,9],
                   "hours_in"    :[1,9,6,5,3,9,2,9,1,7,4,2,2],
                   "happiness"   :[2,1,3,2,3,1,2,3,1,2,2,1,3],
                   "caffienated" :[0,0,1,1,0,0,0,0,1,1,0,1,0]})
```

```
df = pd.DataFrame({"productivity":[5,2,3,1,4,5,6,7,8,3,4,8,9],
                   "hours_in"    :[1,9,6,5,3,9,2,9,1,7,4,2,2],
                   "happiness"   :[2,1,3,2,3,1,2,3,1,2,2,1,3],
                   "caffienated" :[0,0,1,1,0,0,0,0,1,1,0,1,0]})
df.plot.scatter("hours_in", "productivity", s = df["happiness"]*20, c
= df["caffienated"] )  #15min
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb97aef0390>
```



58. What if we want to plot multiple things? Pandas allows you to pass in a matplotlib *Axis* object for plots, and plots will also return an Axis object.

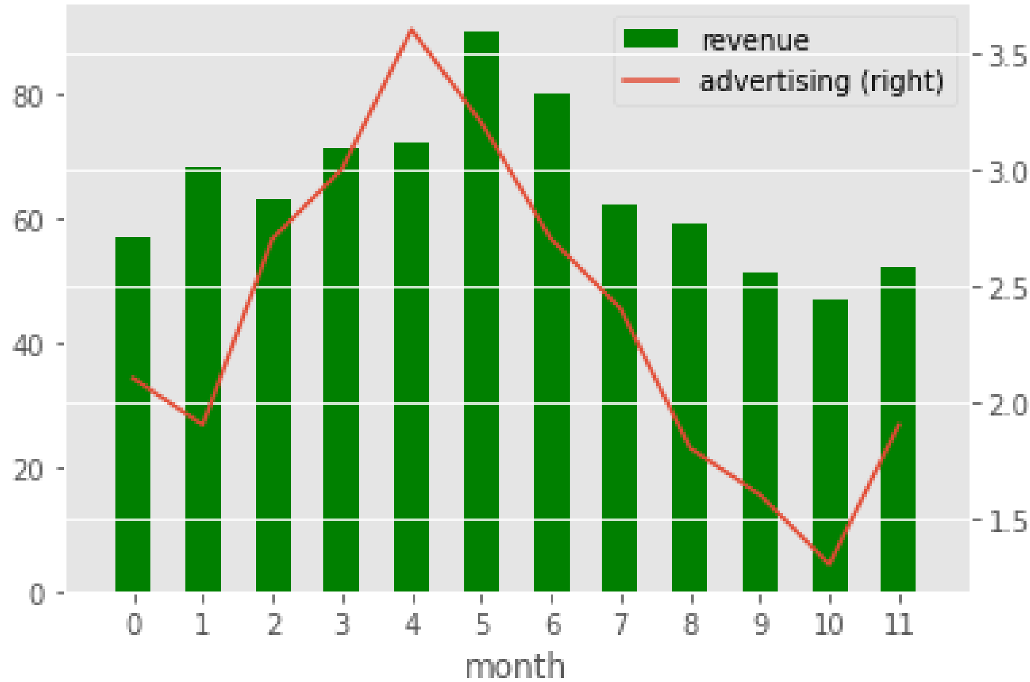
Make a bar plot of monthly revenue with a line plot of monthly advertising spending (numbers in millions)

```
df = pd.DataFrame({"revenue":[57,68,63,71,72,90,80,62,59,51,47,52],
                    "advertising":
[2.1,1.9,2.7,3.0,3.6,3.2,2.7,2.4,1.8,1.6,1.3,1.9],
                    "month":range(12)
                   })
```

```
df = pd.DataFrame({"revenue":[57,68,63,71,72,90,80,62,59,51,47,52],
                    "advertising":
[2.1,1.9,2.7,3.0,3.6,3.2,2.7,2.4,1.8,1.6,1.3,1.9],
                    "month":range(12)
                   })

ax = df.plot.bar("month", "revenue", color = "green")
df.plot.line("month", "advertising", secondary_y = True, ax = ax)
ax.set_xlim((-1, 12))   #15min
```

```
(-1.0, 12.0)
```



Now we're finally ready to create a candlestick chart, which is a very common tool used to analyze stock price data. A candlestick chart shows the opening, closing, highest, and lowest price for a stock during a time window. The color of the "candle" (the thick part of the bar) is green if the stock closed above its opening price, or red if below.

Candlestick Example

This was initially designed to be a pandas plotting challenge, but it just so happens that this type of plot is just not feasible using pandas' methods. If you are unfamiliar with matplotlib, we have provided a function that will plot the chart for you so long as you can use pandas to get the data into the correct format.

Your first step should be to get the data in the correct format using pandas' time-series grouping function. We would like each candle to represent an hour's worth of data. You can write your own aggregation function which returns the open/high/low/close, but pandas has a built-in which also does this.

The below cell contains helper functions. Call `day_stock_data()` to generate a DataFrame containing the prices a hypothetical stock sold for, and the time the sale occurred. Call `plot_candlestick(df)` on your properly aggregated and formatted stock data to print the candlestick chart.

```
import numpy as np
def float_to_time(x):
    return str(int(x)) + ":" + str(int(x%1 * 60)).zfill(2) + ":" +
str(int(x*60 % 1 * 60)).zfill(2)

def day_stock_data():
```

```
    #NYSE is open from 9:30 to 4:00
    time = 9.5
    price = 100
    results = [(float_to_time(time), price)]
    while time < 16:
        elapsed = np.random.exponential(.001)
        time += elapsed
        if time > 16:
            break
        price_diff = np.random.uniform(.999, 1.001)
        price *= price_diff
        results.append((float_to_time(time), price))


    df = pd.DataFrame(results, columns = ['time','price'])
    df.time = pd.to_datetime(df.time)
    return df

#Don't read me unless you get stuck!
def plot_candlestick(agg):
    """
    agg is a DataFrame which has a DatetimeIndex and five columns:
["open","high","low","close","color"]
    """
    fig, ax = plt.subplots()
    for time in agg.index:
        ax.plot([time.hour] * 2, agg.loc[time, ["high","low"]].values,
color = "black")
        ax.plot([time.hour] * 2, agg.loc[time,
["open","close"]].values, color = agg.loc[time, "color"], linewidth =
10)

    ax.set_xlim((8,16))
    ax.set_ylabel("Price")
    ax.set_xlabel("Hour")
    ax.set_title("OHLC of Stock Value During Trading Day")
    plt.show()
```

59. Generate a day's worth of random stock data, and aggregate / reformat it so that it has hourly summaries of the opening, highest, lowest, and closing prices

```
df = day_stock_data()
df.head()

df.set_index("time", inplace = True)
agg = df.resample("H").ohlc()
agg.columns = agg.columns.droplevel()
agg["color"] = (agg.close > agg.open).map({True:"green",False:"red"})
agg.head() #30min
```

```
                         open        high         low       close
color
time

2022-05-30 09:00:00  100.000000  101.152079  99.399094  100.679088
green
2022-05-30 10:00:00  100.590830  101.082190  97.545361   97.956964
red
2022-05-30 11:00:00   97.948237   98.348030  96.099787   97.100671
red
2022-05-30 12:00:00   97.044675   98.393450  95.493610   97.510688
green
2022-05-30 13:00:00   97.477409  100.029392  97.089258   99.648462
green
```

60. Now that you have your properly-formatted data, try to plot it yourself as a candlestick chart. Use the `plot_candlestick(df)` function above, or matplotlib's `plot` documentation if you get stuck.

```
plot_candlestick(agg) #10min
```