Fix a small typo in question 18

6 contributors

1688 lines (1688 sloc) | 51 KB

100 pandas puzzles

Inspired by 100 Numpy exerises, here are 100* 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!

** the list of exercises is not yet complete! Pull requests or suggestions for additional exercises, corrections and improvements are welcomed.*

Importing pandas

Getting started and checking your pandas setup

Difficulty: *easy*

1. Import pandas under the alias `pd`.

In [ ]:
```
import pandas as pd
```

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

In [ ]:
```
pd.__version__
```

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

```
In [ ]:
pd.show_versions()
```

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` .

```
In [ ]:
import numpy as np

data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', '
        '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',

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

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

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*).

```
In [ ]:
df.info()

# ...or...

df.describe()
```

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

In [ ]:

```
df.iloc[:3]

# or equivalently

df.head(3)
```

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

In [ ]:

```
df.loc[:, ['animal', 'age']]

# or

df[['animal', 'age']]
```

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

In [ ]:

```
df.loc[df.index[[3, 4, 8]], ['animal', 'age']]
```

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

In [ ]:

```
df[df['visits'] > 3]
```

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

In [ ]:

```
df[df['age'].isnull()]
```

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

In [ ]:

```
df[(df['animal'] == 'cat') & (df['age'] < 3)]
```

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

In [ ]:

```
df[df['age'].between(2, 4)]
```

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

In [ ]:

```
df.loc['f', 'age'] = 1.5
```

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

In [ ]:

```
df['visits'].sum()
```

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

In [ ]:

```
df.groupby('animal')['age'].mean()
```

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.

In [ ]:

```
df.loc['k'] = [5.5, 'dog', 'no', 2]

# and then deleting the new row...

df = df.drop('k')
```

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

In [ ]:

```
df['animal'].value_counts()
```

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).

In [ ]:

```
df.sort_values(by=['age', 'visits'], ascending=[False, True])
```

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`.

In [ ]:

```
df['priority'] = df['priority'].map({'yes': True, 'no': False})
```

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

In [ ]:

```
df['animal'] = df['animal'].replace('snake', 'python')
```

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*).

In [ ]:

```
df.pivot_table(index='animal', columns='visits', values='age', aggfunc='m
```

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
```

In [ ]:

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

df.loc[df['A'].shift() != df['A']]

# Alternatively, we could use drop_duplicates() here. Note
# that this removes *all* duplicates though, so it won't
# work as desired if A is [1, 1, 2, 2, 1, 1] for example.

df.drop_duplicates(subset='A')
```

23. Given a DataFrame of random numeric values:

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

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

In [ ]:

```
df = pd.DataFrame(np.random.random(size=(5, 3)))

df.sub(df.mean(axis=1), axis=0)
```

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.

In [ ]:

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

df.sum().idxmin()
```

25. How do you count how many unique rows a DataFrame has (i.e. ignore all rows that are duplicates)?

In [ ]:

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

len(df) - df.duplicated(keep=False).sum()

# or perhaps more simply...

len(df.drop_duplicates(keep=False))
```

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.

floating point numbers. Exactly 3 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`

In [ ]:

```
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)


(df.isnull().cumsum(axis=1) == 3).idxmax(axis=1)
```

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
```

In [ ]:

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

df.groupby('grps')['vals'].nlargest(3).sum(level=0)
```

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
```

In [ ]:
```
df = pd.DataFrame(np.random.RandomState(8765).randint(1, 101, size=(100,

df.groupby(pd.cut(df['A'], np.arange(0, 101, 10)))['B'].sum()
```

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'.

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

izero = np.r_[-1, (df == 0).values.nonzero()[0]]  # indices of zeros
idx = np.arange(len(df))
y = df['X'] != 0
df['Y'] = idx - izero[np.searchsorted(izero - 1, idx) - 1]

# http://stackoverflow.com/questions/30730981/how-to-count-distance-to-th
# credit: Behzad Nouri
```

Here's an alternative approach based on a cookbook recipe:

In [ ]:
```
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()
```

And another approach using a groupby operation:

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

df['Y'] = df.groupby((df['X'] == 0).cumsum()).cumcount()
```

```
# We're off by one before we reach the first zero.
first_zero_idx = (df['X'] == 0).idxmax()
df['Y'].iloc[0:first_zero_idx] += 1
```

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)]
```

In [ ]:

```
df = pd.DataFrame(np.random.RandomState(30).randint(1, 101, size=(8, 8)))

df.unstack().sort_values()[-3:].index.tolist()

# http://stackoverflow.com/questions/14941261/index-and-column-for-the-ma
# credit: DSM
```

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
```

In [ ]:

```
df = pd.DataFrame({"vals": np.random.RandomState(31).randint(-30, 30, siz
                   "grps": np.random.RandomState(31).choice(["A", "B"], 1

def replace(group):
    mask = group<0
    group[mask] = group[~mask].mean()
    return group

df.groupby(['grps'])['vals'].transform(replace)
```

```
# http://stackoverflow.com/questions/14760757/replacing-values-with-group
# credit: unutbu
```

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)

In [ ]:

```
df = pd.DataFrame({'group': list('aabbabbbabab'),
                   'value': [1, 2, 3, np.nan, 2, 3, np.nan, 1, 7, 3, np.n

g1 = df.groupby(['group'])['value']              # group values
g2 = df.fillna(0).groupby(['group'])['value']    # fillna, then group val

s = g2.rolling(3, min_periods=1).sum() / g1.rolling(3, min_periods=1).cou

s.reset_index(level=0, drop=True).sort_index()  # drop/sort index

# http://stackoverflow.com/questions/36988123/pandas-groupby-and-rolling-
```

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`.

In [ ]:
```
dti = pd.date_range(start='2015-01-01', end='2015-12-31', freq='B')
s = pd.Series(np.random.rand(len(dti)), index=dti)
s
```

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

In [ ]:
```
s[s.index.weekday == 2].sum()
```

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

In [ ]:
```
s.resample('M').mean()
```

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

In [ ]:
```
s.groupby(pd.Grouper(freq='4M')).idxmax()
```

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

In [ ]:
```
pd.date_range('2015-01-01', '2016-12-31', 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).

In [ ]:

```
df = pd.DataFrame({'From_To': ['LoNDon_paris', 'MAdrid_miLAN', 'londON_St
                               '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 A
                               '12. Air France', '"Swiss Air"']})

df['FlightNumber'] = df['FlightNumber'].interpolate().astype(int)
df
```

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.

In [ ]:

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

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".)

In [ ]:

```
temp['From'] = temp['From'].str.capitalize()
temp['To'] = temp['To'].str.capitalize()
temp
```

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

In [ ]:

```
df = df.drop('From_To', axis=1)
df = df.join(temp)
df
```

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'`.

In [ ]:

```
df['Airline'] = df['Airline'].str.extract('([a-zA-Z\s]+)', expand=False).
# note: using .strip() gets rid of any leading/trailing spaces
df
```

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 new DataFrame named 'delays', rename the columns 'delay_1', 'delay_2', etc. and replace the unwanted RecentDelays column in `df` with 'delays'.

In [ ]:

```
# there are several ways to do this, but the following approach is possib

delays = df['RecentDelays'].apply(pd.Series)

delays.columns = ['delay_{}'.format(n) for n in range(1, len(delays.colum

df = df.drop('RecentDelays', axis=1).join(delays)

df
```

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` .

In [ ]:

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

mi = pd.MultiIndex.from_product([letters, numbers])
s = pd.Series(np.random.rand(30), index=mi)
s
```

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

In [ ]:

```
s.index.is_lexsorted()

# or more verbosely...
s.index.lexsort_depth == s.index.nlevels
```

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

In [ ]:

```
s.loc[:, [1, 3, 6]]
```

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

In [ ]:

```
s.loc[pd.IndexSlice[:'B', 5:]]
```

```
s.loc[pd.IndexSlice[:'B', 5:]]

# or equivalently without IndexSlice...
s.loc[slice(None, 'B'), slice(5, None)]
```

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).

In [ ]:

```
s.sum(level=0)
```

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)` ?

In [ ]:

```
# One way is to use .unstack()...
# This method should convince you that s is essentially just a regular Da
s.unstack().sum(axis=0)
```

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.

In [ ]:

```
new_s = s.swaplevel(0, 1)

if not new_s.index.is_lexsorted():
    new_s = new_s.sort_index()

new_s
```

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
...
```

In [ ]:

```
X = 5
Y = 4

p = pd.core.reshape.util.cartesian_product([np.arange(X), np.arange(Y)])
df = pd.DataFrame(np.asarray(p).T, columns=['x', 'y'])
df
```

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.

In [ ]:

```
# One way is to draw samples from a binomial distribution.

df['mine'] = np.random.binomial(1, 0.4, X*Y)
df
```

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)`.)

In [ ]:

```
# Here is one way to solve using merges.
# It's not necessary the optimal way, just
# the solution I thought of first...

df['adjacent'] = \
    df.merge(df + [ 1,  1, 0], on=['x', 'y'], how='left')\
      .merge(df + [ 1, -1, 0], on=['x', 'y'], how='left')\
      .merge(df + [-1,  1, 0], on=['x', 'y'], how='left')\
      .merge(df + [-1, -1, 0], on=['x', 'y'], how='left')\
      .merge(df + [ 1,  0, 0], on=['x', 'y'], how='left')\
      .merge(df + [-1,  0, 0], on=['x', 'y'], how='left')\
      .merge(df + [ 0,  1, 0], on=['x', 'y'], how='left')\
      .merge(df + [ 0, -1, 0], on=['x', 'y'], how='left')\
       .iloc[:, 3:]\
        .sum(axis=1)

# An alternative solution is to pivot the DataFrame
# to form the "actual" grid of mines and use convolution.
# See https://github.com/jakevdp/matplotlib_pydata2013/blob/master/exampl

from scipy.signal import convolve2d
```

```
mine_grid = df.pivot_table(columns='x', index='y', values='mine')
counts = convolve2d(mine_grid.astype(complex), np.ones((3, 3)), mode='sam
df['adjacent'] = (counts - mine_grid).ravel('F')
```

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

In [ ]:

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

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

In [ ]:

```
df.drop('mine', axis=1).set_index(['y', 'x']).unstack()
```

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!

In [ ]:

```
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")
```

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