# /var/

## Various programming stuff

- Css
- Django
- Flask
- Git
- Javascript
- Pelican
- Postgresql
- Python
- Spring
- Wagtail

## A pandas pivot_table primer

Τετ 21 Σεπτέμβριος 2016

### Introduction

Recently, I started using the pandas python library to improve the quality (and quantity) of statistics in my applications. One pandas method that I use frequently and is really powerful is `pivot_table`. This is a rather complex method that has very poor documentation. Beyond this, this command is explained a little more in an article about data reshaping, however, even this leaves much to be desired (when I first tried reading it I was overwhelmed by the amount of information there).

A great introduction to pandas is the three part series by Greg Reda - it touches `pivot_table` however I was only able to understand it properly *after* I played a lot with it. I don't know, maybe playing with `pivot_table` yourself (or being really experienced in such concepts) is the only way to properly comprehend it! To help with this journey however I'm going to try to explain various basic pandas concepts that will lead us to the `pivot_table` command (and some of its friends). Notice that I'm using pandas 0.18.1.

I have to mention that I am no expert in statistics or numeric analysis so this post won't have any advanced information and may even point out some obvious things. However keep in mind things that may seem obvious to some experts are really difficult to grasp for a non-expert.

Before continuing, please notice that this article has been written as a jupyter notebook and was integrated with pelican using the pelican-ipynb plugin. I had to do some modifications to my theme to better integrate the notebook styling, however some stuff may not look as nice as the other articles. I have to mention that this integration is really great and I totally recommend it!

## The DataFrame

The most important data structure that pandas uses is the DataFrame. This can be thought as a two dimensional array, something like an Excel spreadsheet. In the pandas nomenclature, the rows of that two-dimensional array are called *indexes* (while the columns are still called *columns*) — I'll either use rows or indexes for the rows of the DataFrame. The rows are called indexes because they can be used to … index data (think of each column as a dictionary). However please notice that pandas has a different data structure named Index that is used to store the names of the headers (axis) of the rows and columns.

If we horizontally/vertically pick the values of a single row(index)/column we'll be left with a different data structure called Series - this is more or less a single dimensional array (or a dictionary with the names of the columns/indexes as keys). There's also a Panel data structure which is 3-dimensional, more like a complete Excel workbook (the third dimension being the individual sheets of the workbook) but I won't cover that here.

More info on the above can be found on the corresponding article about data structures.

There are various ways to read the data for a Series or DataFrame: Initializing through arrays or dicts, reading from csv, xls, database, combinining series to create an array and various others. I won't go into any details about this but will include some examples on how to create Series and DataFrames. If you are familiar with python you can just convert everything to a dict and read that instead of researching individual methods.

## Using Series

The `Series` data structure is more or less used to store a single dimensional array of data. This array-like structure could either have numbers as indexes (so will be more similar to a normal array) or have textual indexes (so will be more similar to a dictionary). Let's see some examples:

In [203]:
```
import pandas as pd

def t(o):
    # Return the class name of the object
    return o.__class__.__name__

# Use an array to create a Series
series1 = pd.Series([10,20,30])
print "series1 (", t(series1), ')\n', series1
# Notice that the index names were automatically generated as 0,1,2
# Use a dict to create a Series
# notice that the keys of the dict will be the index names
series2 = pd.Series({'row1':11,'row2':22,'row3':33})
print "series2 (", t(series2), ')\n', series2
```

```
series1 ( Series )
0    10
1    20
2    30
dtype: int64
series2 ( Series )
row1    11
row2    22
row3    33
dtype: int64
```

The are various ways to select values from the Series. You can use textual or numeric indexes or you can filter the elements using an intuitive syntax:

In [204]:
```
# Get values from series using named indexes
series2['row1']
# Can also use slicing and interesting operations
# like array in array [[]] to select specific indexes
print series1[1:]
print series1[[0,2]]
print series2['row2':]
print series2[['row1', 'row3']]
```

```
1    20
2    30
dtype: int64
0    10
2    30
dtype: int64
row2    22
row3    33
dtype: int64
row1    11
row3    33
dtype: int64
```

In [205]:
```
# Filtering series
# You can use comparison operators with a Series to
# get an array of booleans with the result of each element
print "Boolean result\n", series2>15
# This boolean array can then be used to filter the Series
# by returning only the elements that are "True"
print "Filtered result\n", series2[series2>15]
```

```
Boolean result
row1    False
row2     True
row3     True
dtype: bool
Filtered result
row2    22
row3    33
dtype: int64
```

In [206]:
```
# The above means that we'll only get the second and third (index: 0,2)

# So we can create a function that returns Boolean, apply it to
# all elements of series with map and use the result for indexing!
def is_22(x):
    return x==22
print "Map filtering\n", series2[series2.map(is_22)]
```

```
Map filtering
row2    22
dtype: int64
```

The `map` method above gets a callback function and applies it to all elements of the Series, returning a new Series with the results. It is similar to the `map(function, sequence) -> list` global python funtion. Using map filtering is the most general way to filter elements of a series.

## Using DataFrames

Let’s start by a quick introduction to see some basic operations on DataFrames:

In [207]:

```
# Create a DataFrame using a two-dimensional array
# Notice that the indexes and column names were automatically generated
df1 = pd.DataFrame([[10,20,30], [40,50,60]])
print "Dataframe from array: df1(", t(df1), ')'
print df1

# Use a dict to give names to columns
df2 = pd.DataFrame([{'col1':10,'col2':20,'col3':30}, {'col1':40,'col2':
print "Dataframe from dict: df2(", t(df2), ')'
print df2

# Give names to indexes
df3 = pd.DataFrame([
    {'col1':10,'col2':20,'col3':30},
    {'col1':40,'col2':50,'col3':60}
], index=['idx1', 'idx2'])
print "Dataframe from dict, named indexes: df3(", t(df3), ')'
print df3

# What happens when columns are missing
df4 = pd.DataFrame([{'col1':10,'col2':20,'col3':30}, {'col2':40,'col3':
print "Dataframe from dict, missing columns: df4(", t(df4), ')'
print df4

# Create a DataFrame by combining series
df5 = pd.DataFrame([pd.Series([1,2]), pd.Series([3,4])], index=['a', 'b
print "Dataframe from series: df5(", t(df5), ')'
print df5

# Output a dataframe as html
print df5.to_html()

# Notice that there are many more interesting DataFrame output methods,
# to_csv, to_dict, to_excel, to_json, to_latex, to_msgpack, to_string,
```

```
Dataframe from array: df1( DataFrame )
    0   1   2
0  10  20  30
1  40  50  60
Dataframe from dict: df2( DataFrame )
   col1  col2  col3
0    10    20    30
1    40    50    60
Dataframe from dict, named indexes: df3( DataFrame )
      col1  col2  col3
idx1    10    20    30
idx2    40    50    60
Dataframe from dict, missing columns: df4( DataFrame )
   col1  col2  col3  col4
0  10.0    20    30   NaN
1   NaN    40    50  60.0
Dataframe from series: df5( DataFrame )
   0  1
a  1  2
b  3  4
```

| | **0** | **1** |
|---|---|---|
| **a** | 1 | 2 |
| **b** | 3 | 4 |

Reading a DataFrame from an array of python dicts is (at least for me) the easiest way to put my data in a DataFrame. Use any normal python method to generate that array of dicts and then just initialize the DataFrme with that. Also, the `to_html` method is really useful to quickly output a DataFrame to your web application - don't forget to add some styling to the `.dataframe` class!

Selecting values from the Dataframe is very easy if you know how to do it. You index (`[]`) directly to select columns:

In [208]:
```
print "df3(", t(df3), ")\n", df3

# We can get a column as a Series
print "Get column as series\n", df3['col3']
# Or multiple columns as a DataFrame
print "Get multiple columns\n", df3[['col3', 'col2']]

# We can also get the column by its idx
print "Get column by index\n", df3[df3.columns[1]]

# Pick values from a dataframe using array indexing
# df3['col2'] returns a Series so using the ['idx2']
# index to it will return the actual value
print "Get value\n", df3['col2']['idx2']
```

```
df3( DataFrame )
      col1  col2  col3
idx1    10    20    30
idx2    40    50    60
Get column as series
idx1    30
idx2    60
Name: col3, dtype: int64
Get multiple columns
      col3  col2
idx1    30    20
idx2    60    50
Get column by index
idx1    20
idx2    50
Name: col2, dtype: int64
Get value
50
```

Also you use the `loc/iloc` properties of the `DataFrame` to select rows/indexes (either by number or by text). The `loc/iloc` actually behave as a two dimensional array - they can get two parameters, the first one being the row/rows and the second one being the column/columns:

In [209]:
```
# Pick an index (select a horizontal line) as a series
print "Get index as a series\n", df3.loc['idx1']
# Also can pick by index number
print "Get index as a series by index\n", df3.iloc[0]
# iloc can be used to numerically index both rows and columns by passin
print "Two dimensional - get by index\n",df3.iloc[0, :] # This is the s
# so to select the first column we'll use
print "Two dimensional - get by column\n", df3.iloc[:, 0]
# We could do more interesting things, for example select a square
print "Two dimensional - get by index and column\n", df3.iloc[0:2, 1:3]
# Loc which is for label based indexing can also be used as a two dimen
print "Two dimensional - use label based indexing\n", df3.loc[['idx1','
```

```
Get index as a series
col1    10
col2    20
col3    30
Name: idx1, dtype: int64
Get index as a series by index
col1    10
col2    20
col3    30
Name: idx1, dtype: int64
Two dimensional - get by index
col1    10
col2    20
col3    30
Name: idx1, dtype: int64
Two dimensional - get by column
idx1    10
idx2    40
Name: col1, dtype: int64
Two dimensional - get by index and column
      col2  col3
idx1    20    30
idx2    50    60
Two dimensional - use label based indexing
idx1    10
idx2    40
Name: col1, dtype: int64
```

Of course, boolean indexing and filtering can also be used just like in Series:

In [210]:
```
print "Boolean dataframe\n", df3>30
print "Boolean indexing\n",df3[df3>30]
def is_20_or_50(x):
    return x==20 or x==50
# We need to use applymap (instead of map we used in Series)
print "Boolean indexing\n",df3[df3.applymap(is_20_or_50)]
```

```
Boolean dataframe
       col1   col2   col3
idx1  False  False  False
idx2   True   True   True
Boolean indexing
      col1  col2  col3
idx1   NaN   NaN   NaN
idx2  40.0  50.0  60.0
Boolean indexing
      col1  col2  col3
idx1   NaN    20   NaN
idx2   NaN    50   NaN
```

Notice that for the DataFrame we use the `applymap` method which applies the callback function to all individual elements of the DataFrame and returns the result as a new DataFrame (with the same dimensions of course). The boolean indexing is nice but it does not actually drop not needed things, we see that we just get a NaN in the positions that are filtered. Could we do something better? The answer is yes, but we'll need to do index/column boolean indexing - i.e select only specific columns or indexes and then pass these to filter the dataframe:

In [211]:
```
# Let's see the indexes that have *10* in their col1 column
print df3['col1']==10
# And then select *only* these indexes (i.e idx1)
print df3[df3['col1']==10]
# Now we can do exactly the opposite (see columns that have 10 in their
print df3.loc['idx1']==10
# And then select *only* these columns (i.e col1)
print df3.loc[:, df3.loc['idx1']==10]
```

```
idx1     True
idx2    False
Name: col1, dtype: bool
      col1  col2  col3
idx1    10    20    30
col1     True
col2    False
col3    False
Name: idx1, dtype: bool
      col1
idx1    10
idx2    40
```

In [212]:
```
# Let's finally see a general solution to boolean selecting with loc:
# Select specific columns
print df3.loc[:, [True, False, True] ]
# Select specific rows
print df3.loc[[False, True], : ]
# Select specific rows and cols
print df3.loc[[False, True], [True, False,True] , ]
# So we can pass two boolean arrays to loc, the first for selecting ind
# the second for selecting columns
```

```
      col1  col3
idx1    10    30
idx2    40    60
      col1  col2  col3
idx2    40    50    60
      col1  col3
idx2    40    60
```

## Modifying DataFrames

It’s easy to modify the DataFrame by changing its values, adding more indexes / columns, dropping rows and columns, renaming columns and indexes. Notice that some operations are performed in place (so they modify the original DataFrame), while others return a copy of the original array.

In [213]:
```
# Let's copy because some of the following operators change the datafra
df = df3.copy()
print df

print "Change values of a column"
df['col1'] = [11,41]
print df

print "Change values of an index"
df.loc['idx1'] = [11,21, 31]
print df

print "We can change more specific values (a 2x2 array here)"
df.iloc[0:2, 0:2] = [[4,3], [2,1]]
print df

print "Add another column to an existing dataframe (changes DataFrame)"
df['col4'] = [1,2]
print df

print "Add another row (index) to an existing dataframe (changes DataFr
df.loc['idx3']=[100,200,300,400]
print df

print "Drop a row (returns new object)"
print df.drop('idx1')

print "Drop a column (returns new object)"
print df.drop('col1', axis=1)

print "Rename index (returns new object)"
print df.rename(index={'idx1': 'new-idx-1'})

print "Rename column (returns new object)"
print df.rename(columns={'col1': 'new-col-1'})

print "Transpose array- change columns to rows and vice versa"
print df.T

print "Double transpose - returns the initial DataFrame"
print df.T.T
```

```
      col1  col2  col3
idx1    10    20    30
idx2    40    50    60
Change values of a column
      col1  col2  col3
idx1    11    20    30
idx2    41    50    60
Change values of an index
      col1  col2  col3
idx1    11    21    31
idx2    41    50    60
We can change more specific values (a 2x2 array here)
      col1  col2  col3
idx1     4     3    31
idx2     2     1    60
Add another column to an existing dataframe (changes DataFrame)
      col1  col2  col3  col4
idx1     4     3    31     1
idx2     2     1    60     2
Add another row (index) to an existing dataframe (changes DataFrame)
      col1  col2  col3  col4
idx1     4     3    31     1
idx2     2     1    60     2
```

## More advanced operations

Beyond the previous, more or less basic operations, pandas allows you to do some advanced operations like SQL-like joins of more than one dataset or, applying a function to each of the rows / columns or even individual cells of the DataFrame:

In [214]:
```
authors_df=pd.DataFrame([{'id': 1, 'name':'Stephen King'}, {'id': 2, 'n

books_df=pd.DataFrame([
    {'id': 1, 'author_id':1, 'name':'It'},
    {'id': 2, 'author_id':1, 'name':'The Stand'},
    {'id': 3, 'author_id':2, 'name':'Airframe'},
    {'id': 4, 'author_id':2, 'name':'Jurassic Park'}
])

print authors_df
print books_df
print books_df.merge(authors_df, left_on='author_id', right_on='id')
```

```
   id             name
0   1     Stephen King
1   2  Michael Crichton
   author_id  id           name
0          1   1             It
1          1   2      The Stand
2          2   3       Airframe
3          2   4  Jurassic Park
   author_id  id_x         name_x  id_y            name_y
0          1     1             It     1      Stephen King
1          1     2      The Stand     1      Stephen King
2          2     3       Airframe     2  Michael Crichton
3          2     4  Jurassic Park     2  Michael Crichton
```

As can be seen above, the `merge` method of DataFrame can be used to do an sql-like join with another DataFrame, using specific columns as join-keys for each of the two dataframes (`left_on` and `right_on`). There are a lot of options for doing various join types (left, right, inner, outer etc) and concatenating DataFrames with other ways - most are discussed in the corresponding post.

Let's see another method of doing the above join that is more controlled, using the `apply` method of DataFrame that *applies* a function to each row/column of the DataFrame and returns the result as a series:

In [215]:
```
# Let's do the join using a different method
def f(r):
    author_df_partial = authors_df[authors_df['id']==r['author_id']]
    return author_df_partial.iloc[0]['name']

books_df['author name'] = books_df.apply(f, axis=1)
print books_df
```

```
   author_id  id           name       author name
0          1   1             It      Stephen King
1          1   2      The Stand      Stephen King
2          2   3       Airframe  Michael Crichton
3          2   4  Jurassic Park  Michael Crichton
```

How does this work? We pass the `axis=1` parameter to `apply` so that the callback function will be called for each row of the DataFrame (by default `axis=0` which means it will be called for each column). So, `f` will be called getting each row as an input. From this `book_df` row, we get the `author_id` it contains and filter `authors_df` by it. Notice that `author_df_partial` is actually a DataFrame containing only one row, so we need to filter it by getting its only line, using `iloc[0]` which will return a Series and finally, we return the author name using the corresponding index `name`.

When calling the `apply` method, by defautl the `axis` parameter is 0 (i.e the function will be called for each column). When I first encountered this I found it very strange because I thought that most users would usually want to apply a function to each of the rows. However, there's a reason for applying the function to all columns, here's an example:

In [216]:
```
values = pd.DataFrame([
    {'temperature': 31, 'moisture': 68},
    {'temperature': 33, 'moisture': 72},
    {'temperature': 31.5, 'moisture': 58},
    {'temperature': 28.5, 'moisture': 42},
])

import numpy as np
# We can easily create statistics for our data using apply -- that's wh
# axis=0 is the default parameter to apply (to operate vertically to ea
values.loc['avg']=values.apply(np.average )
values.loc['len']=values.apply(len )
values.loc['sum']=values.apply(sum)
print values
```

```
     moisture  temperature
0        68.0         31.0
1        72.0         33.0
2        58.0         31.5
3        42.0         28.5
avg      60.0         31.0
len       5.0          5.0
sum     305.0        160.0
```

## Comprehending `pivot_table`

After this (rather long) introduction to using and manipulating DataFrames, the time has come to see `pivot_table`. The `pivot_table` method is applied to a DataFrame and its purpose is to "reshape" and "aggregate" the values of a DataFrame . More on reshaping can be found here and it means changing the indexes/columns of the DataFrame to create a new DataFrame that fits our needs. Aggregate on the other hand means that for each of the cells of the new DataFrame we'll create a summary of the data that should have appeared there.

Let's start by creating a nice set of data we'll use for the `pivot_table` operations:

The recommended type of input (at least by me) to the `pivot_table` is a simple DataFrame like the one I have already created: Your index will be the `id` of your database (or you could even have an auto-generated index like in the example) and the columns will be the values you want to aggregate and reshape. This is very easy to create either by reading a file (xls/csv) or by a simple SQL query (substituting all foreign keys with a representative value). In the above example, we actually have the following columns: *author, genre, name, pages, year, decade, size* - this is a pool of data that will be very useful to remember for later and it is important to also keep it in your mind for your data. So, use a unique id as the index and remember the names of your columns.

As we can see in the documentation, the [pivot_table method]() uses four basic parameters:

- `index`: An array of the data that will be used as indexes to the resulting (i.e the reshaped and aggregated) DataFrame
- `columns`: An array of the data that will be used as a columns to the resulting DataFrame
- `values`: An array of the data whose values we want to aggregate in each cell
- `aggfunc`: Which is the function (or functions) that will be used for aggregating the values

So, how it actually works? You select a number of the headers from your pool of data and assign them to either `index` or `columns`, depending if you want to put them horizontally or vertically. Notice that both `index` and `columns`:

- take either a string (to denote a single column) or an array to denote multiple columns
- are optional (but you must define one of them) — if you skip either columns or index you'll get a `Series` instead of a `DataFrame`
- are interchangable (you can put any header from your pool to either index or columns, depending on how you want to display your data)
- are mutually exclusive (you can't put the same header in both index and columns)

Multiple data headers means that you'll have [hierachical indexes / columns]() in your pivot (or `MultiIndex` as it's called - remember that `Index` is used to store the axis of the DataFrame), ie the rows/columns would be grouped by a hierarchy. Let's see an example of multiple indexes:

If we used `'decade'` as an index, then the pivot_table index would be like

- `70s` value1 value2 …
- `80s` value1 value2 …
- `90s` value1 value2 …

while, if we used `['decade', 'year']` we'd hove something like

- `70s`
    - `1975` value1 value2 …
    - `1978` value1 value2 …
- `80s`
    - `1980` value1 value2 …
    - `1982` value1 value2 …
    - …
- `90s`
    - `1990` value1 value2 …
    - …

So, each year would automatically be grouped to its corresponing decade. The same would be true if we used `['decade', 'year']` in columns (but we'll now have a vertical grouping from top to bottom). Notice that pandas doesn't know if

In [218]:
```
books_df=pd.DataFrame([
    {'author':'Stephen King', 'name':'It', 'pages': 1138, 'year': 1986,
    {'author':'Stephen King', 'name':'The Stand', 'pages': 823, 'year':
    {'author':'Stephen King', 'name': 'Salem\'s Lot', 'pages': 439, 'ye
    {'author':'Stephen King', 'name': 'Misery', 'pages': 320, 'year': 1
    {'author':'Stephen King', 'name': 'Pet Sematary', 'pages': 374, 'ye
    {'author':'Stephen King', 'name': 'Bag of bones', 'pages': 529, 'ye
    {'author':'Stephen King', 'name': 'Different Seasons', 'pages': 527
    {'author':'Stephen King', 'name': 'The Dark Tower: The Gunslinger',
    {'author':'Stephen King', 'name': 'The Dark Tower II: The Drawing o
    {'author':'Stephen King', 'name': 'The Dark Tower III: The Waste La
    {'author':'Stephen King', 'name': 'The Dark Tower IV: Wizard and Gl
    {'author':'Michael Crichton', 'name':'Airframe', 'pages': 352, 'yea
    {'author':'Michael Crichton', 'name':'Jurassic Park', 'pages': 448,
    {'author':'Michael Crichton', 'name':'Congo', 'pages': 348, 'year':
    {'author':'Michael Crichton', 'name':'Sphere', 'pages': 385, 'year'
    {'author':'Michael Crichton', 'name':'Rising Sun', 'pages': 385, 'y
    {'author':'Michael Crichton', 'name':'Disclosure ', 'pages': 597, '
    {'author':'Michael Crichton', 'name':'The Lost World ', 'pages': 43
    {'author':'John Grisham', 'name':'A Time to Kill', 'pages': 515, 'y
    {'author':'John Grisham', 'name':'The Firm', 'pages': 432, 'year':1
    {'author':'John Grisham', 'name':'The Pelican Brief', 'pages': 387,
    {'author':'John Grisham', 'name':'The Chamber', 'pages': 496, 'year
    {'author':'John Grisham', 'name':'The Rainmaker', 'pages': 434, 'ye
    {'author':'John Grisham', 'name':'The Runaway Jury', 'pages': 414,
    {'author':'John Grisham', 'name':'The Street Lawyer', 'pages': 347,
    {'author':'George Pelecanos', 'name':'Nick\'s Trip ', 'pages': 276,
    {'author':'George Pelecanos', 'name':'A Firing Offense', 'pages': 2
    {'author':'George Pelecanos', 'name':'The Big Blowdown', 'pages': 3
    {'author':'George R.R Martin', 'name':'A Clash of Kings', 'pages':
    {'author':'George R.R Martin', 'name':'A Game of Thrones', 'pages':
])

# Add a decade column to the books DataFrame
def add_decade(y):
        return str(y['year'])[2] + '0\'s'

books_df['decade'] = books_df.apply(add_decade, axis=1)

# Add a size column to the books DataFrame
def add_size(y):
        if y['pages'] > 600:
            return 'big'
        elif y['pages'] < 300:
            return 'small'
        return 'medium'

books_df['size'] = books_df.apply(add_size, axis=1)
# Let's display it sorted here
books_df.sort_values(['decade', 'genre', 'year'])
```

Out[218]:

| | author | genre | name | pages | year | decade | size |
|---|---|---|---|---|---|---|---|
| **2** | Stephen King | Horror | Salem’s Lot | 439 | 1975 | 70’s | medium |
| **1** | Stephen King | Horror | The Stand | 823 | 1978 | 70’s | big |
| **18** | John Grisham | Crime | A Time to Kill | 515 | 1989 | 80’s | medium |
| **13** | Michael Crichton | Fantasy | Congo | 348 | 1980 | 80’s | medium |
| **7** | Stephen King | Fantasy | The Dark Tower: The Gunslinger | 224 | 1982 | 80’s | small |

In [219]:
```
# Here's the first example
books_df.pivot_table(index=['decade', ], columns=['genre'], )
```

Out[219]:

| | pages | | | | year | | |
|---|---|---|---|---|---|---|---|
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Crime** | **Fantasy** | **Horr** |
| **decade** | | | | | | | |
| **70's** | NaN | NaN | 631.0 | NaN | NaN | NaN | 1976. |
| **80's** | 515.000000 | 339.25 | 756.0 | 423.5 | 1989.000000 | 1984.000000 | 1984. |
| **90's** | 387.416667 | 606.50 | 529.0 | NaN | 1994.083333 | 1994.666667 | 1998. |

In the above, we aggregated dour books by their decade and genre.

As we can see we just passed `decade` as an index and `genre` as a column. We ommited `values` and `aggfunc` so the default values were used. What happened? Pandas created a new DataFrame that had the values of decade as its index and the values of genre as its columns. Now, for each of the values (remember that since we ommited values, pandas just gets all numerical data, i.e pages and year) it found the corresponding entries for each cell, got their average and put it in that cell. For example, since there are no Crime genre books in the 70's we got a `NaN` to both the pages and year values. However, there are two Horror books, with 823 and 439 pages so their average is 631. Notice that for each value a separate top-level multi-column containing all indexes and columns was created - we can display only pages or year by indexing with `['pages']` or `['year']`. We can think of each of the values columns as a seperate pivot table, so in the above example we have a pivot table for pages and a pivot table for year.

The above year column will also use the default average aggregate, something that doesn't actually makes sense. So we can use `values` to explicitly define which values to aggregate — here's how we can display only the pages:

In [220]:
```
books_df.pivot_table(index=['decade', ], columns=['genre'], values='pag
#The above is more or less the same as with books_df.pivot_table(index=
```

Out[220]:

| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** |
|---|---|---|---|---|
| **decade** | | | | |
| **70's** | NaN | NaN | 631.0 | NaN |
| **80's** | 515.000000 | 339.25 | 756.0 | 423.5 |
| **90's** | 387.416667 | 606.50 | 529.0 | NaN |

In [221]:
```
# In the above, we could pass ['pages'] instead of 'pages' as the value
# This will result in creating a multi-column index with 'pages' as the
books_df.pivot_table(index=['decade', ], columns=['genre'], values=['pa
```

Out[221]:

| | pages | | | |
|---|---|---|---|---|
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** |
| **decade** | | | | |
| **70's** | NaN | NaN | 631.0 | NaN |
| **80's** | 515.000000 | 339.25 | 756.0 | 423.5 |
| **90's** | 387.416667 | 606.50 | 529.0 | NaN |

In [222]:
```
# Also, please notice that you can skip index or columns (but not both)
print books_df.pivot_table(index=['decade', ], values='pages')
print books_df.pivot_table(columns=['decade', ], values='pages')
```

```
decade
70's    631.000000
80's    470.111111
90's    464.052632
Name: pages, dtype: float64
decade
70's    631.000000
80's    470.111111
90's    464.052632
Name: pages, dtype: float64
```

Notice that above we have exactly the same result since for both cases we got a `Series` (it doesn't matter that we used index in the first and columns in the second). Also, since we use *less* columns from our data pool (we used only decade while previously we used both decade and genre), the aggregation is more coarse: We got the averages of book pages in each decade. Of course, we could have the same values as before but use a multi-column index:

In [223]:
```
s1 = books_df.pivot_table(index=['decade', 'genre'], values='pages')
s2 = books_df.pivot_table(columns=['decade', 'genre'], values='pages')
print "s1 equals s2: ", s1.equals(s2)
print s1.index
s1
```

```
s1 equals s2:  True
MultiIndex(levels=[[u'70's', u'80's', u'90's'], [u'Crime', u'Fantasy', u
           labels=[[0, 1, 1, 1, 1, 2, 2, 2], [2, 0, 1, 2, 3, 0, 1, 2]],
           names=[u'decade', u'genre'])
```

Out[223]:
```
decade  genre
70's    Horror      631.000000
80's    Crime       515.000000
        Fantasy     339.250000
        Horror      756.000000
        Thriller    423.500000
90's    Crime       387.416667
        Fantasy     606.500000
        Horror      529.000000
Name: pages, dtype: float64
```

The above return a Series with a multi column index (they are both the same). Notice that the data is exactly the same as when we passed decade and genre in in index and column. The only difference is that some NaN rows have been dropped from the Series while in the DataFrame are there, for example Crime/70's (the DataFrame will by default drop a row or index if all its values are NaN). Finally, take a look at how the multi index is represented (each easy to decypher it).

Let's now say that we actually wanted to have a meaningful value for the year, for example the first year we have a book for that genre/decade:

In [224]:
```
# I'll intentionally skip values again to see what happens
books_df.pivot_table(index=['decade', ], columns=['genre'], aggfunc=min
```

Out[224]:

| | **author** | | | | **name** | | | |
|---|---|---|---|---|---|---|---|---|
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Crime** | **Fantasy** | **Horror** | **Thrille** |
| **decade** | | | | | | | | |
| **70's** | None | None | Stephen King | None | None | None | Salem's Lot | None |
| **80's** | John Grisham | Michael Crichton | Stephen King | Stephen King | A Time to Kill | Congo | It | Differen Seasons |
| **90's** | George Pelecanos | George R.R Martin | Stephen King | None | A Firing Offense | A Clash of Kings | Bag of bones | None |

This is more interesting. It seems that since we didn't use the default aggfunc value but instead we passed our own (`min`), pandas did not use only the numerical values but used instead *all remaining columns* as values: Remember that our pool of data was *author, genre, name, pages, year, decade, size*, the genre and decade were used as an index/column so the remaining headers were used as values: author, name, pages, year, size! For the pages and year we can understand what happens: For example, for the Horror novels of the 80's, the one with the minimal pages is Pet Sematery with 374 pages. The same has also the minimal year (1983). However, the one with the minimal name is `It` (since `I` is before `P` it just compares strings). The author is the same for both(`Stephen King`) and the minimum size is medium (since `small (s) > medium (m)`). Of course we could pass the `values` parameter to actually define which values we wanted to see.

Another really interesting thing is to take a peek at which are the values that are passed to the aggregation function. For this, we can just use `tuple`:

In [225]:
```
# Please notice that for reasons unknown to me, if I used aggfunc=tuple
books_df_tuples = books_df.pivot_table(index=['decade', ], columns=['ge
books_df_tuples
```

Out[225]:

| | author | | | | name | | |
|---|---|---|---|---|---|---|---|
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Crime** | **Fantasy** | **Horr** |
| **decade** | | | | | | | |
| **70's** | None | None | (Stephen King, Stephen King) | None | None | None | (The Stand, Salem': Lot) |
| **80's** | (John Grisham,) | (Stephen King, Stephen King, Michael Crichton,... | (Stephen King, Stephen King) | (Stephen King, Stephen King) | (A Time to Kill,) | (The Dark Tower: The Gunslinger, The Dark Towe... | (It, Pet Semata |
| **90's** | (Michael Crichton, Michael Crichton, Michael C... | (Stephen King, Stephen King, Michael Crichton,... | (Stephen King,) | None | (Airframe, Rising Sun, Disclosure, The Firm, ... | (The Dark Tower III: The Waste Lands, The Dark... | (Bag of bones,) |

In [226]:
```
# Dont worry about the ellipsis, the values are all there in each cell,
books_df_tuples['author']['Crime']['90\'s']
```

Out[226]:
```
('Michael Crichton',
 'Michael Crichton',
 'Michael Crichton',
 'John Grisham',
 'John Grisham',
 'John Grisham',
 'John Grisham',
 'John Grisham',
 'John Grisham',
 'George Pelecanos',
 'George Pelecanos',
 'George Pelecanos')
```

Notice that for the columns we havea MultiIndex of both value_type (author, name etc) and genre (Crime, Fantasy etc) while for the index we have the decade. So by `books_df_tuples['author']` we'll get the author values DataFrame, by `books_df_tuples['author']['Crime']` we'll get the Crime column of that DataFrame as a series and finally with `books_df_tuples['author']['Crime']['90\'s']` we'll get the actuall value which is all author names that have written Crime books in the 90's — authors that have written multiple books will be displayed multiple times.

What if we wanted to only display the different authors for each genre and decade and remove duplicates:

In [227]:
```
books_df.pivot_table(
    index=['decade', ],
    columns=['genre'],
    values='author',
    aggfunc=lambda x: ', '.join(set(x))
)
```

Out[227]:

| genre | Crime | Fantasy | Horror | Thriller |
|---|---|---|---|---|
| decade | | | | |
| 70's | None | None | Stephen King | None |
| 80's | John Grisham | Stephen King, Michael Crichton | Stephen King | Stephen King |
| 90's | John Grisham, Michael Crichton, George Pelecanos | Stephen King, George R.R Martin, Michael Crichton | Stephen King | None |

What happens above is that we use the `lambda x: ', '.join(set(x))` function to aggregate. This function will create a set (i.e remove duplicates) from the input (which is the corresponding values for each cell) and then join the set members using `','`.

Notice that the inpout parameter that is passed to our aggfunc is actually a `Series` so don't be alarmed if some list operations are not working:

In [228]:
```
books_df.pivot_table(
    index=['decade', ],
    columns=['genre'],
    values='author',
    aggfunc=lambda x: type(x)
)
```

Out[228]:

| genre | Crime | Fantasy | Horror | Thriller |
|---|---|---|---|---|
| decade | | | | |
| 70's | None | None | | None |
| 80's | | | | |
| 90's | | | | None |

Before continuing, I'd like to present another two parameters that could be passed to the pivot_table: `fill_value` to define a value to display when no values are found to be aggregated for a cell and `margins` to enable or disable margin rows/columns to the left/bottom that will aggregate all values of that column, for example:

In [229]:
```
books_df.pivot_table(
    index=['decade', ],
    columns=['genre'],
    values ='author',
    aggfunc=lambda x: ', '.join(set(x)),
    margins=True,
    fill_value='-'
)
```

Out[229]:

| genre | Crime | Fantasy | Horror | Thriller | All |
|---|---|---|---|---|---|
| decade | | | | | |
| 70's | - | - | Stephen King | - | Stephen King |
| 80's | John Grisham | Stephen King, Michael Crichton | Stephen King | Stephen King | Stephen King, John Grisham, Michael Crichton |
| 90's | John Grisham, Michael Crichton, George Pelecanos | Stephen King, George R.R Martin, Michael Crichton | Stephen King | - | Stephen King, George R.R Martin, John Grisham,... |
| All | John Grisham, Michael Crichton, George Pelecanos | Stephen King, George R.R Martin, Michael Crichton | Stephen King | Stephen King | Stephen King, George R.R Martin, John Grisham,... |

The "All" column above will aggregate all values for each row/column (and the All/All down right will aggregate all values).

Using our previous knowledge of multi column indexes, let's display the average number of pages each author writes for each decade and genre:

In [230]:
```
books_df.pivot_table(
    index=['decade', ],
    columns=['author', 'genre'],
    values='pages',
)
```

Out[230]:

| author | George Pelecanos | George R.R Martin | John Grisham | Michael Crichton | | Stephen K | |
|---|---|---|---|---|---|---|---|
| genre | Crime | Fantasy | Crime | Crime | Fantasy | Fantasy | Horror |
| decade | | | | | | | |
| 70's | NaN | NaN | NaN | NaN | NaN | NaN | 631.0 |
| 80's | NaN | NaN | 515.000000 | NaN | 366.5 | 312.0 | 756.0 |
| 90's | 268.333333 | 731.0 | 418.333333 | 444.666667 | 439.0 | 649.5 | 529.0 |

In [231]:
```
# One interesting thing is that if we changed the order of the multi-co
books_df.pivot_table(
    index=['decade', ],
    columns=['genre', 'author'],
    values='pages',
)
```

Out[231]:

| genre | Crime | | | Fantasy | | | Horr |
|---|---|---|---|---|---|---|---|
| author | George Pelecanos | John Grisham | Michael Crichton | George R.R Martin | Michael Crichton | Stephen King | Steph Kin |
| decade | | | | | | | |
| 70's | NaN | NaN | NaN | NaN | NaN | NaN | 631.0 |
| 80's | NaN | 515.000000 | NaN | NaN | 366.5 | 312.0 | 756.0 |
| 90's | 268.333333 | 418.333333 | 444.666667 | 731.0 | 439.0 | 649.5 | 529.0 |

In [232]:
```
# Or we can interchange index with columns to get the same data in a ho
books_df.pivot_table(
    columns=['decade', ],
    index=['author', 'genre'],
    values='pages',
)
```

Out[232]:

| | decade | 70's | 80's | 90's |
|---|---|---|---|---|
| author | genre | | | |
| George Pelecanos | Crime | NaN | NaN | 268.333333 |
| George R.R Martin | Fantasy | NaN | NaN | 731.000000 |
| John Grisham | Crime | NaN | 515.0 | 418.333333 |
| Michael Crichton | Crime | NaN | NaN | 444.666667 |
| | Fantasy | NaN | 366.5 | 439.000000 |
| Stephen King | Fantasy | NaN | 312.0 | 649.500000 |
| | Horror | 631.0 | 756.0 | 529.000000 |
| | Thriller | NaN | 423.5 | NaN |

So, `Michael Crichton` was writing 445 pages for Crime novels and 439 pages for Fantasy novels on average at the 90's (of course this would be true if we had included all works of Michael Crichton). In the previous table we can see that, for example for `George Pelecanos` only the `Crime` genre is displayed (since he's only Crime genre books in our database). Pandas automatically drops columns / lines where everything is empty (NaN)— if we for some reason wanted to display it, could use the `dropna=False` parameter:

In [233]:
```
books_df.pivot_table(
    index=['decade', ],
    columns=['author', 'genre'],
    values=['pages'],
    dropna=False
)
```

Out[233]:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **author** | **George Pelecanos** | | | | **George R.R Martin** | | | |
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Crime** | **Fantasy** | **Horror** | **Thriller** |
| **decade** | | | | | | | | |
| **70's** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **80's** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **90's** | 268.333333 | NaN | NaN | NaN | NaN | 731.0 | NaN | NaN |

In [234]:
```
# We can create any combination we want with our multi-index colums, fo
# be decade / year / author and genre / size
books_df.pivot_table(
    index=['decade', 'year', 'author', 'name' ],
    columns=['size', 'genre'],
    values='pages',
    aggfunc=lambda x: 'v',
    fill_value='',
)
```

Out[234]:

| | | | size | big | | medium | | |
|---|---|---|---|---|---|---|---|---|
| | | | genre | Fantasy | Horror | Crime | Fantasy | Horror |
| decade | year | author | name | | | | | |
| 70's | 1975 | Stephen King | Salem's Lot | | | | | v |
| | 1978 | Stephen King | The Stand | | v | | | |
| 80's | 1980 | Michael Crichton | Congo | | | | v | |
| | 1982 | Stephen King | Different Seasons | | | | | |
| | | | The Dark Tower: The Gunslinger | | | | | |
| | 1983 | Stephen King | Pet Sematary | | | | | v |
| | 1986 | Stephen King | It | | v | | | |
| | 1987 | Michael Crichton | Sphere | | | | v | |
| | | Stephen King | Misery | | | | | |
| | | | The Dark Tower II: The Drawing of the Three | | | | v | |
| | 1989 | John Grisham | A Time to Kill | | | v | | |
| 90's | 1990 | Michael Crichton | Jurassic Park | | | | v | |
| | 1991 | John Grisham | The Firm | | | v | | |
| | | Stephen King | The Dark Tower III: The Waste Lands | | | | v | |
| | 1992 | George Pelecanos | A Firing Offense | | | | | |
| | | John Grisham | The Pelican Brief | | | v | | |
| | | Michael Crichton | Rising Sun | | | v | | |
| | 1993 | George Pelecanos | Nick's Trip | | | | | |

One more advanced thing I'd like to cover here is that we could define multiple aggregate functions for each one of our values by passing a dictionary of `value:function` to the `aggfunc` parameter. For example, if we wanted to display

- the sum of the pages that have been written
- the range of years for which we have books
- the names of the authors
- the name of one book we have

for each genre each decade, we could do something like this

In [235]:

```
def get_range(years):
    return '{0} - {1}'.format(min(years), max(years))

def get_names(authors):
    return ', '.join(set(authors))

def get_book(books):
    # Don't forget the the passed parameter is a Series so we use iloc
    return books.iloc[0]

books_df.pivot_table(
    index=['decade',  ],
    columns=['genre', ],
    values=['author', 'pages', 'year', 'name'],
    aggfunc={
        'author': get_names,
        'pages': sum,
        'year': get_range,
        'name': get_book,
    },
    fill_value='-'
)
```

Out[235]:

| | year | | | | pages | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Crime** | **Fantasy** | **Horror** | **Thriller** | **Cr** |
| **decade** | | | | | | | | | |
| **70's** | - | - | 1975 - 1978 | - | - | - | 1262 | - | - |
| **80's** | 1989 - 1989 | 1980 - 1987 | 1983 - 1986 | 1982 - 1987 | 515 | 1357 | 1512 | 847 | A T to K |
| **90's** | 1991 - 1998 | 1990 - 1998 | 1998 - 1998 | - | 4649 | 3639 | 529 | - | Airf |

As we've already mentioned, the above is more or less like four different pivot tables — for example we could get a pivot table with only pages if we passed `'pages'` as the `values` and `sum` as the `aggfunc` in the above method call.

## Friends of pivot_table

The `pivot_table` method has some friends — these are functions that operate on DataFrame and can do reshaping but they are not as powerful as pivot_table. Let's introduce some of them:

In [236]:
```
# First, I'll create a DataFrame as an example:
df=books_df.pivot_table(index=['decade', ], columns=['genre', 'author']
# This df has a Multi-index in columns - first level is the genres, sec
print df.columns
print df.index
df
```

```
MultiIndex(levels=[[u'Crime', u'Fantasy', u'Horror', u'Thriller'], [u'G
           labels=[[0, 0, 0, 1, 1, 1, 2, 3], [0, 2, 3, 1, 3, 4, 4, 4]],
           names=[u'genre', u'author'])
Index([u'70's', u'80's', u'90's'], dtype='object', name=u'decade')
```

Out[236]:

| genre | Crime | | | Fantasy | | | Horror | T |
|---|---|---|---|---|---|---|---|---|
| author | George Pelecanos | John Grisham | Michael Crichton | George R.R Martin | Michael Crichton | Stephen King | Stephen King | S |
| decade | | | | | | | | |
| 70's | NaN | NaN | NaN | NaN | NaN | NaN | 1262.0 | N |
| 80's | NaN | 515.0 | NaN | NaN | 733.0 | 624.0 | 1512.0 | 8 |
| 90's | 805.0 | 2510.0 | 1334.0 | 1462.0 | 878.0 | 1299.0 | 529.0 | N |

Notice above the `MultiIndex` and `Index` structs that are used to hold the axis for columns and index.

## Stack / unstack

These two operations move columns to indexes and vice-versa. Let's see what the manual says:

- stack: Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.
- unstack: Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

I must confess that I was not able to comprehend the above! A more easy explanation is that:

- stack will re-arrange the values of the DataFrame so that the most inner column (the one at the bottom) will be converted to the most inner index (to the right)
- unstack will do the exactly opposite: Re-arrange the values of the DataFrame so that the most inner index (the one at the right) will be converted to the most inner column (to the bottom)

Also, stack and unstack do not really make sense. It would be much easier (at least to me) if stack was named `col_to_idx` (or `col_to_row`) and unstack was named `idx_to_col` (`row_to_col`).

Before looking at examples of stack and unstack let's take a look at the index and columns of our dataframe. Notice again the `Index` and `MultiIndex` data structs:

In [237]:
```
print "Index\n",df.index
print "Column\n",df.columns
```

```
Index
Index([u'70's', u'80's', u'90's'], dtype='object', name=u'decade')
Column
MultiIndex(levels=[[u'Crime', u'Fantasy', u'Horror', u'Thriller'], [u'G
           labels=[[0, 0, 0, 1, 1, 1, 2, 3], [0, 2, 3, 1, 3, 4, 4, 4]],
           names=[u'genre', u'author'])
```

In [238]:
```
stacked = df.stack()
print "Index\n",stacked.index
print "Column\n",stacked.columns
stacked
```

```
Index
MultiIndex(levels=[[u'70's', u'80's', u'90's'], [u'George Pelecanos', u
           labels=[[0, 1, 1, 1, 2, 2, 2, 2, 2], [4, 2, 3, 4, 0, 1, 2, 3
           names=[u'decade', u'author'])
Column
Index([u'Crime', u'Fantasy', u'Horror', u'Thriller'], dtype='object', na
```

Out[238]:

| | **genre** | **Crime** | **Fantasy** | **Horror** | **Thriller** |
|---|---|---|---|---|---|
| **decade** | **author** | | | | |
| **70's** | **Stephen King** | NaN | NaN | 1262.0 | NaN |
| **80's** | **John Grisham** | 515.0 | NaN | NaN | NaN |
| | **Michael Crichton** | NaN | 733.0 | NaN | NaN |
| | **Stephen King** | NaN | 624.0 | 1512.0 | 847.0 |
| **90's** | **George Pelecanos** | 805.0 | NaN | NaN | NaN |
| | **George R.R Martin** | NaN | 1462.0 | NaN | NaN |
| | **John Grisham** | 2510.0 | NaN | NaN | NaN |
| | **Michael Crichton** | 1334.0 | 878.0 | NaN | NaN |
| | **Stephen King** | NaN | 1299.0 | 529.0 | NaN |

We see that the `author` column (which was the most inner column) was moved to the right of the indexes. The rows (index) was converted to a multi-index while the columns is a simple index now.

In [239]:
```
# We can of course stack again -- this time we'll get a series (with a
stacked2 = stacked.stack()
print stacked2.index
stacked2
```

```
MultiIndex(levels=[[u'70's', u'80's', u'90's'], [u'George Pelecanos', u
           labels=[[0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2], [4, 2, 3, 4
           names=[u'decade', u'author', u'genre'])
```

Out[239]:
```
decade  author             genre
70's    Stephen King       Horror      1262.0
80's    John Grisham       Crime        515.0
        Michael Crichton   Fantasy      733.0
        Stephen King       Fantasy      624.0
                           Horror      1512.0
                           Thriller     847.0
90's    George Pelecanos   Crime        805.0
        George R.R Martin  Fantasy     1462.0
        John Grisham       Crime       2510.0
        Michael Crichton   Crime       1334.0
                           Fantasy      878.0
        Stephen King       Fantasy     1299.0
                           Horror       529.0
dtype: float64
```

In [240]:
```
# unstack does the opposite operation
unstacked = df.unstack()
print unstacked.index
unstacked
```

```
MultiIndex(levels=[[u'Crime', u'Fantasy', u'Horror', u'Thriller'], [u'Ge
           labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1
           names=[u'genre', u'author', u'decade'])
```

Out[240]:
```
genre     author              decade
Crime     George Pelecanos    70's         NaN
                              80's         NaN
                              90's       805.0
          John Grisham        70's         NaN
                              80's       515.0
                              90's      2510.0
          Michael Crichton    70's         NaN
                              80's         NaN
                              90's      1334.0
Fantasy   George R.R Martin   70's         NaN
                              80's         NaN
                              90's      1462.0
          Michael Crichton    70's         NaN
                              80's       733.0
                              90's       878.0
          Stephen King        70's         NaN
                              80's       624.0
                              90's      1299.0
Horror    Stephen King        70's      1262.0
                              80's      1512.0
                              90's       529.0
Thriller  Stephen King        70's         NaN
                              80's       847.0
                              90's         NaN
dtype: float64
```

We now see that the that the `decade` column (which was the only index) was moved as the most inner to the columns — however this also converts this DataFrame to a Series!

One interesting thing to notice is that a Series can only be `unstack()`ed since it has no columns (so `stack` won't work, remember stack = col_to_idx)

In [241]:
```
# unstack - move the rightmost idx (decade) to columns
unstacked.unstack()
```

Out[241]:

| | decade | 70's | 80's | 90's |
|---|---|---|---|---|
| genre | author | | | |
| Crime | George Pelecanos | NaN | NaN | 805.0 |
| | John Grisham | NaN | 515.0 | 2510.0 |
| | Michael Crichton | NaN | NaN | 1334.0 |
| Fantasy | George R.R Martin | NaN | NaN | 1462.0 |
| | Michael Crichton | NaN | 733.0 | 878.0 |
| | Stephen King | NaN | 624.0 | 1299.0 |
| Horror | Stephen King | 1262.0 | 1512.0 | 529.0 |
| Thriller | Stephen King | NaN | 847.0 | NaN |

In [243]:
```
#Also, because `unstack` works on series we can use it for ever to cycl
df.unstack()
df.unstack().unstack()
df.unstack().unstack().unstack().unstack().unstack().unstack().unstack(
```

Out[243]:

| decade | 70's | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| author | George Pelecanos | George R.R Martin | John Grisham | Michael Crichton | Stephen King | George Pelecanos | George R.R Martin | ( |
| genre | | | | | | | | |
| Crime | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 5 |
| Fantasy | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 1 |
| Horror | NaN | NaN | NaN | NaN | 1262.0 | NaN | NaN | 1 |
| Thriller | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 1 |

In [244]:
```
# One final comment is that stack and unstack can get a level parameter
# index/column level we want to pivot
# For example the following will unstack - idx_to_col the leftmost inde
unstacked.unstack(level=0)
```

Out[244]:

| | genre | Crime | Fantasy | Horror | Thriller |
|---|---|---|---|---|---|
| author | decade | | | | |
| George Pelecanos | 70's | NaN | NaN | NaN | NaN |
| | 80's | NaN | NaN | NaN | NaN |
| | 90's | 805.0 | NaN | NaN | NaN |
| George R.R Martin | 70's | NaN | NaN | NaN | NaN |
| | 80's | NaN | NaN | NaN | NaN |
| | 90's | NaN | 1462.0 | NaN | NaN |
| John Grisham | 70's | NaN | NaN | NaN | NaN |
| | 80's | 515.0 | NaN | NaN | NaN |
| | 90's | 2510.0 | NaN | NaN | NaN |
| Michael Crichton | 70's | NaN | NaN | NaN | NaN |
| | 80's | NaN | 733.0 | NaN | NaN |
| | 90's | 1334.0 | 878.0 | NaN | NaN |
| Stephen King | 70's | NaN | NaN | 1262.0 | NaN |
| | 80's | NaN | 624.0 | 1512.0 | 847.0 |
| | 90's | NaN | 1299.0 | 529.0 | NaN |

## pivot

The pivot command will convert a column values to an index. This is similar like the `pivot_table` but does not aggregate the values and does not create multi-hierarchy indexes so you must be careful that each cell will contain only one value.

In [245]:
```
# We'll use the initial books_df DataFrame
books_df.pivot(index='name', columns='genre', values='year')
# Notice that we used 'name' as an index (to be sure that each cell wil
```

Out[245]:

| genre | Crime | Fantasy | Horror | Thriller |
|---|---|---|---|---|
| **name** | | | | |
| **A Clash of Kings** | NaN | 1998.0 | NaN | NaN |
| **A Firing Offense** | 1992.0 | NaN | NaN | NaN |
| **A Game of Thrones** | NaN | 1996.0 | NaN | NaN |
| **A Time to Kill** | 1989.0 | NaN | NaN | NaN |
| **Airframe** | 1996.0 | NaN | NaN | NaN |
| **Bag of bones** | NaN | NaN | 1998.0 | NaN |
| **Congo** | NaN | 1980.0 | NaN | NaN |
| **Different Seasons** | NaN | NaN | NaN | 1982.0 |
| **Disclosure** | 1994.0 | NaN | NaN | NaN |
| **It** | NaN | NaN | 1986.0 | NaN |
| **Jurassic Park** | NaN | 1990.0 | NaN | NaN |
| **Misery** | NaN | NaN | NaN | 1987.0 |
| **Nick's Trip** | 1993.0 | NaN | NaN | NaN |
| **Pet Sematary** | NaN | NaN | 1983.0 | NaN |
| **Rising Sun** | 1992.0 | NaN | NaN | NaN |
| **Salem's Lot** | NaN | NaN | 1975.0 | NaN |
| **Sphere** | NaN | 1987.0 | NaN | NaN |
| **The Big Blowdown** | 1996.0 | NaN | NaN | NaN |
| **The Chamber** | 1994.0 | NaN | NaN | NaN |
| **The Dark Tower II: The Drawing of the Three** | NaN | 1987.0 | NaN | NaN |
| **The Dark Tower III: The Waste Lands** | NaN | 1991.0 | NaN | NaN |
| **The Dark Tower IV: Wizard and Glass** | NaN | 1998.0 | NaN | NaN |
| **The Dark Tower: The Gunslinger** | NaN | 1982.0 | NaN | NaN |
| **The Firm** | 1991.0 | NaN | NaN | NaN |
| **The Lost World** | NaN | 1995.0 | NaN | NaN |
| **The Pelican Brief** | 1992.0 | NaN | NaN | NaN |
| **The Rainmaker** | 1995.0 | NaN | NaN | NaN |
| **The Runaway Jury** | 1996.0 | NaN | NaN | NaN |
| **The Stand** | NaN | NaN | 1978.0 | NaN |
| **The Street Lawyer** | 1998.0 | NaN | NaN | NaN |

In [246]:
```
# We could pivot by using name as a column
books_df.pivot(index='decade', columns='name', values='pages')
```

Out[246]:

| name | A Clash of Kings | A Firing Offense | A Game of Thrones | A Time to Kill | Airframe | Bag of bones | Congo | Different Seasons | Dis |
|---|---|---|---|---|---|---|---|---|---|
| **decade** | | | | | | | | | |
| **70's** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | Nal |
| **80's** | NaN | NaN | NaN | 515.0 | NaN | NaN | 348.0 | 527.0 | Nal |
| **90's** | 768.0 | 216.0 | 694.0 | NaN | 352.0 | 529.0 | NaN | NaN | 597 |

3 rows × 30 columns

What happens is that we got the values of one column and coverted these to a column/index and use another column's values as the values of the new DataFrame. So, in the first example the values of the genre column were converted to columns and inside each cell we put the page number. In the second example instead we converted the decade value to index and put the page number inside each cell. In both cases we used the name of the book to be sure that we would each cell will contain one value (remember that pivot cannot aggregate).

The `pivot` command is not very useful (at least to me) since it does not actually modify (by aggregating) data but just changes its representation - you won't get any new information from `pivot` but you'll only display it differently. Also keep in mind that each cell after the pivoting must contain *one* value, for example in the above wouldn't work if we used `author` as columns (instead of book name).

## groupby

The final method we'll talk about and is related to `pivot_table` is [`groupby`](). This of course is related to the SQL group by method and should be easy to comprehend. The `groupby` gets a parameter that defines how to group the entries and returns a GroupBy object that contains the groups. The GroupBy object can be enumerated to get the groups and their data. It's interesting to take a look at the structure of each such object:

In [247]:
```
groupby_object = books_df.groupby(['decade', 'author'])
print type(groupby_object)
# Let's see what groupby-object contains
for x in groupby_object:
    print "type: ", type(x), "len: ", len(x) #len(x), type(x[0]), type(
    print "first element of tuple", type(x[0]), x[0]
    print "second element of tuple", type(x[1])
```

```
type:   len:  2
first element of tuple  ("70's", 'Stephen King')
second element of tuple
type:   len:  2
first element of tuple  ("80's", 'John Grisham')
second element of tuple
type:   len:  2
first element of tuple  ("80's", 'Michael Crichton')
second element of tuple
type:   len:  2
first element of tuple  ("80's", 'Stephen King')
second element of tuple
type:   len:  2
first element of tuple  ("90's", 'George Pelecanos')
second element of tuple
type:   len:  2
first element of tuple  ("90's", 'George R.R Martin')
second element of tuple
type:   len:  2
first element of tuple  ("90's", 'John Grisham')
second element of tuple
type:   len:  2
first element of tuple  ("90's", 'Michael Crichton')
second element of tuple
type:   len:  2
first element of tuple  ("90's", 'Stephen King')
second element of tuple
```

So, from the above we can see that the GroupBy object contains a number of 2-element tuples. Each tuple contains (another tuple with) the columns that were used for groupping and the actual data of that group (as a DataFrame). Now, we could either use the enumeration I shown above to operate on each group or, better, to use some of the methods that the GroupBy object contains:

In [248]:
```
# get some statistics
print groupby_object.mean()
print groupby_object.sum()

# We can use the aggregate method to do anything we want
# Each aggregate function will get a Series with the values (similar to
def year_aggr(x):
    return '{0}-{1}'.format(max(x), min(x))

def genre_aggr(x):
    return ', '.join(set(x))

groupby_object.aggregate({'year':year_aggr, 'pages': sum, 'genre':genre
```

```
                                  pages         year
decade author
70's   Stephen King        631.000000  1976.500000
80's   John Grisham        515.000000  1989.000000
       Michael Crichton    366.500000  1983.500000
       Stephen King        497.166667  1984.500000
90's   George Pelecanos    268.333333  1993.666667
       George R.R Martin   731.000000  1997.000000
       John Grisham        418.333333  1994.333333
       Michael Crichton    442.400000  1993.400000
       Stephen King        609.333333  1995.666667
                           pages   year
decade author
70's   Stephen King         1262   3953
80's   John Grisham          515   1989
       Michael Crichton      733   3967
       Stephen King         2983  11907
90's   George Pelecanos      805   5981
       George R.R Martin    1462   3994
       John Grisham         2510  11966
       Michael Crichton     2212   9967
       Stephen King         1828   5987
```

Out[248]:

| | | genre | pages | year |
|---|---|---|---|---|
| **decade** | **author** | | | |
| **70's** | **Stephen King** | Horror | 1262 | 1978-1975 |
| **80's** | **John Grisham** | Crime | 515 | 1989-1989 |
| | **Michael Crichton** | Fantasy | 733 | 1987-1980 |
| | **Stephen King** | Fantasy, Horror, Thriller | 2983 | 1987-1982 |
| **90's** | **George Pelecanos** | Crime | 805 | 1996-1992 |
| | **George R.R Martin** | Fantasy | 1462 | 1998-1996 |
| | **John Grisham** | Crime | 2510 | 1998-1991 |
| | **Michael Crichton** | Fantasy, Crime | 2212 | 1996-1990 |
| | **Stephen King** | Fantasy, Horror | 1828 | 1998-1991 |

In [249]:
```
# It's interesting to notice that the previous is exactly
# the same that we can do with this pivot_table command
books_df.pivot_table(index=['decade', 'author'], values=['genre', 'page
    'genre': genre_aggr,
    'year': year_aggr,
    'pages': sum,
})
```

Out[249]:

| | | genre | pages | year |
|---|---|---|---|---|
| **decade** | **author** | | | |
| **70's** | **Stephen King** | Horror | 1262 | 1978-1975 |
| **80's** | **John Grisham** | Crime | 515 | 1989-1989 |
| | **Michael Crichton** | Fantasy | 733 | 1987-1980 |
| | **Stephen King** | Fantasy, Horror, Thriller | 2983 | 1987-1982 |
| **90's** | **George Pelecanos** | Crime | 805 | 1996-1992 |
| | **George R.R Martin** | Fantasy | 1462 | 1998-1996 |
| | **John Grisham** | Crime | 2510 | 1998-1991 |
| | **Michael Crichton** | Fantasy, Crime | 2212 | 1996-1990 |
| | **Stephen King** | Fantasy, Horror | 1828 | 1998-1991 |

In [250]:
```
# The added value of pivot_table over group of course is that we could
books_df.pivot_table(columns=['decade'], index=['author'], values=['gen
        'genre': genre_aggr,
        'year': year_aggr,
        'pages': sum,
    })
# or any other combination of columns - index we wanted
```

Out[250]:

| | genre | | | pages | | | year | | |
|---|---|---|---|---|---|---|---|---|---|
| **decade** | **70's** | **80's** | **90's** | **70's** | **80's** | **90's** | **70's** | **80's** | |
| **author** | | | | | | | | | |
| **George Pelecanos** | None | None | Crime | None | None | 805 | None | None | 19 |
| **George R.R Martin** | None | None | Fantasy | None | None | 1462 | None | None | 19 |
| **John Grisham** | None | Crime | Crime | None | 515 | 2510 | None | 1989-1989 | 19 |
| **Michael Crichton** | None | Fantasy | Fantasy, Crime | None | 733 | 2212 | None | 1987-1980 | 19 |
| **Stephen King** | Horror | Fantasy, Horror, Thriller | Fantasy, Horror | 1262 | 2983 | 1828 | 1978-1975 | 1987-1982 | 19 |

## A real-world example

To continue with a real-world example, I will use the MovieLens 100k to represent some `pivot_table` (and friends) operations. To load the data I've used the code already provided by the three part series I already mentioned. Notice that this won't load the genre of the movie (left as an excersize to the reader).

In [251]:
```
# Useful to display graphs inline
%matplotlib inline
```

In [252]:
```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

path = 'C:/Users/serafeim/Downloads/ml-100k' # Change this to your own
u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
users = pd.read_csv(os.path.join(path, 'u.user'), sep='|', names=u_cols
r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv(os.path.join(path, 'u.data'), sep='\t', names=r_c
m_cols = ['movie_id', 'title', 'release_date', 'video_release_date', 'i
movies = pd.read_csv(os.path.join(path, 'u.item'), sep='|', names=m_col
movie_ratings = movies.merge(ratings)
lens = movie_ratings.merge(users)
lens.head()
```

Out[252]:

| | movie_id | title | release_date | video_release_date | imdb_u |
|---|---|---|---|---|---|
| 0 | 1 | Toy Story (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Toy%20Story%2... |
| 1 | 4 | Get Shorty (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Get%20Shorty%... |
| 2 | 5 | Copycat (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Copycat%20(199 |
| 3 | 7 | Twelve Monkeys (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Twelve%20Monk... |
| 4 | 8 | Babe (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Babe%20(1995) |

As we can see, we are using the `merge` method of DataFrame to do an SQL-style join between `movies` and `ratings` and then between `movie_ratings` and `users` - this will result in a fat DataFrame with all the info of the movie and user for each review. The `head` method displays the 5 first rows of the DataFrame.

We can see that there's a `zip_code` - I wanted to convert it to the specific US-state. There's a service from `ziptasticapi.com` that can be used for that but you need to do the queries one-by-one! I've executed the queries once and created a zip-state dict to be used instead:

In [253]:
```
# The following can be used to find out the state by zip_code for each
API="http://ziptasticapi.com/{0}"
states = {}
import urllib2, json
def get_state(s):
    global states
    if states.get(s):
        return states.get(s)
    headers = { 'User-Agent' : 'Mozilla/5.0' }
    req = urllib2.Request(API.format(s), None, headers)
    state = json.loads(urllib2.urlopen(req).read()).get('state')
    states[s] = state
    return state

#using this command we can add the state column
#lens['state']=lens['zip_code'].apply(get_state)
```

In [254]:
```
# However, since we shouldn't call the zipstatic API so many times, I'l
# dict of zip:state (that I actually got through the previous command)
states2={u'73013': u'OK', u'77042': u'TX', u'61455': u'IL', u'55345': u
u'19716': u'DE', u'55343': u'MN', u'15203': u'PA', u'48446': u'MI', u'9
u'92653': u'CA', u'61073': u'IL', u'55346': u'MN', u'32303': u'FL', u'3
u'32712': u'FL', u'06437': u'CT', u'01581': u'MA', u'85719': u'AZ', u'1
u'10960': u'NY', u'32789': u'FL', u'01375': u'MA', u'60135': u'IL', u'9
u'95521': u'CA', u'49512': u'MI', u'02215': u'MA', u'80209': u'CO', u'9
u'98006': u'WA', u'52302': u'IA', u'60187': u'IL', u'46005': None, u'46
u'63021': u'MO', u'17036': u'PA', u'99206': u'WA', u'10707': u'NY', u'7
u'21208': u'MD', u'75204': u'TX', u'60007': u'IL', u'60005': u'IL', u'2
u'21201': u'MD', u'21206': u'MD', u'22906': u'VA', u'45680': u'OH', u'9
u'53144': u'WI', u'05001': u'VT', u'97208': u'OR', u'54494': u'WI', u'9
u'45660': u'OH', u'53214': u'WI', u'53210': u'WI', u'53211': u'WI', u'3
u'37412': u'TN', u'63119': u'MO', u'10025': u'NY', u'10022': u'NY', u'1
u'44648': u'OH', u'60641': u'IL', u'78213': u'TX', u'78212': u'TX', u'2
u'96819': u'HI', u'42647': u'KY', u'62901': u'IL', u'62903': u'IL', u'9
u'04102': u'ME', u'14627': u'NY', u'20006': u'DC', u'70808': u'LA', u'2
u'20001': u'DC', u'70802': u'LA', u'05452': u'VT', u'20009': u'DC', u'2
u'08610': u'NJ', u'33775': u'FL', u'30329': u'GA', u'76013': u'TX', u'8
u'11758': u'NY', u'95014': u'CA', u'08052': u'NJ', u'37777': u'TN', u'3
u'76309': u'TX', u'23509': u'VA', u'50311': u'IA', u'33884': u'FL', u'3
u'42459': u'KY', u'95064': u'CA', u'02859': u'RI', u'68504': u'NE', u'4
u'68503': u'NE', u'02918': u'RI', u'34656': u'FL', u'L1V3W': None, u'22
u'55113': u'MN', u'55117': u'MN', u'55116': u'MN', u'23112': u'VA', u'9
u'91206': u'CA', u'06927': u'CT', u'55337': u'MN', u'02136': u'MA', u'1
u'47130': u'IN', u'02139': u'MA', u'02138': u'MA', u'N2L5N': None, u'15
u'15213': u'PA', u'50670': u'IA', u'04988': u'ME', u'19382': u'PA', u'8
u'55454': u'MN', u'19149': u'PA', u'19146': u'PA', u'55021': u'MN', u'V
u'06405': u'CT', u'73071': u'OK', u'77459': u'TX', u'92037': u'CA', u'6
u'64118': u'MO', u'21114': u'MD', u'98101': u'WA', u'98103': u'WA', u'9
u'02341': u'MA', u'94306': u'CA', u'94305': u'CA', u'85233': u'AZ', u'1
u'90814': u'CA', u'14534': u'NY', u'98072': u'WA', u'16803': u'PA', u'4
u'10309': u'NY', u'95468': u'CA', u'60402': u'IL', u'60152': u'IL', u'7
u'98199': u'WA', u'12603': u'NY', u'90254': u'CA', u'84116': u'UT', u'1
u'41850': None, u'97214': u'OR', u'97215': u'OR', u'97212': u'OR', u'10
u'10018': u'NY', u'49705': u'MI', u'10011': u'NY', u'10010': u'NY', u'1
u'13210': u'NY', u'78209': u'TX', u'60659': u'IL', u'01754': u'MA', u'6
u'70124': u'LA', u'12345': u'NY', u'95161': u'CA', u'20015': u'DC', u'9
u'58202': u'ND', u'29379': u'SC', u'94703': u'CA', u'94702': u'CA', u'6
u'24060': u'VA', u'33763': u'FL', u'33765': u'FL', u'54248': None, u'80
u'03062': u'NH', u'03060': u'NH', u'18301': u'PA', u'08403': u'NJ', u'9
u'48043': u'MI', u'28450': u'NC', u'78264': u'TX', u'63304': u'MO', u'0
u'08105': u'NJ', u'07102': u'NJ', u'18015': u'PA', u'11231': u'NY', u'2
u'38115': u'TN', u'95076': u'CA', u'77845': u'TX', u'77841': u'TX', u'1
u'08360': u'NJ', u'02903': u'RI', u'01945': u'MA', u'40256': u'KY', u'9
u'89801': u'NV', u'48825': u'MI', u'48823': u'MI', u'07204': u'NJ', u'9
u'55106': u'MN', u'55107': u'MN', u'55104': u'MN', u'55105': u'MN', u'5
u'55109': u'MN', u'61755': u'IL', u'91351': u'CA', u'Y1A6B': None, u'91
u'28734': u'NC', u'55320': u'MN', u'78205': u'TX', u'11201': u'NY', u'0
u'47024': u'IN', u'43212': u'OH', u'43215': u'OH', u'02125': u'MA', u'0
u'15222': u'PA', u'M7A1A': None, u'97520': u'OR', u'76234': u'TX', u'55
u'55423': u'MN', u'55422': u'MN', u'55038': u'MN', u'55428': u'MN', u'9
u'T8H1N': None, u'16125': u'PA', u'02154': None, u'R3T5K': None, u'3580
u'97006': u'OR', u'02159': None, u'32250': u'FL', u'50613': u'IA', u'92
u'21044': u'MD', u'98117': u'WA', u'E2A4H': None, u'90804': u'CA', u'74
u'22903': u'VA', u'22904': u'VA', u'52245': u'IA', u'52246': u'IA', u'5
u'17331': u'PA', u'20723': u'MD', u'63044': u'MO', u'17110': u'PA', u'1
u'32605': u'FL', u'60067': u'IL', u'90247': u'CA', u'61820': u'IL', u'8
u'84105': u'UT', u'84107': u'UT', u'60090': u'IL', u'99835': u'AK', u'9
u'05201': u'VT', u'10003': u'NY', u'20090': u'DC', u'90064': u'CA', u'0
u'21250': u'MD', u'20657': u'MD', u'97203': u'OR', u'60466': u'IL', u'4
u'44134': u'OH', u'78390': u'TX', u'44133': u'OH', u'83686': u'ID', u'1
u'45810': u'OH', u'75006': u'TX', u'63146': u'MO', u'91335': u'CA', u'3
```

Beyond the state, I'd like to add some other columns for describing data and drop a bunch of non-needed columns:

In [255]:
```
import datetime

# Let's also initialize it by the release_year, decade and review day
lens['release_year']=lens['release_date'].apply(lambda x: str(x).split(
lens['decade']=lens['release_year'].apply(lambda x: str(x)[2:3]+"0's" i
lens['review_day']=lens['unix_timestamp'].apply(lambda x: datetime.date

# And remove some non-needed stuff
final_lens = lens.drop(['release_date','zip_code', 'unix_timestamp', 'v

# Also add an idx column
final_lens['idx'] = final_lens.index

final_lens.head()
```

Out[255]:

| | title | rating | age | sex | occupation | state | release_year | decade | review_da |
|---|---|---|---|---|---|---|---|---|---|
| **0** | Toy Story (1995) | 4 | 60 | M | retired | CA | 1995 | 90's | Tuesday |
| **1** | Get Shorty (1995) | 5 | 60 | M | retired | CA | 1995 | 90's | Tuesday |
| **2** | Copycat (1995) | 4 | 60 | M | retired | CA | 1995 | 90's | Tuesday |
| **3** | Twelve Monkeys (1995) | 4 | 60 | M | retired | CA | 1995 | 90's | Tuesday |
| **4** | Babe (1995) | 5 | 60 | M | retired | CA | 1995 | 90's | Tuesday |

So, after the previous (I hope easy) modifications we have a DataFrame that contains useful info about reviews of movies. Each line of the dataframe contains the following 9 columns of data:

- Title of movie
- Rating it got from this review
- Age of the reviewer
- Sex of the reviewer
- Occupation of the reviewer
- State (US) of the reviewer
- Release year of the movie
- Decade the movie was released
- Day of the review

Let's take a peek at the movies of which decade were prefered by reviewers, by sex. First of all, we'll do a pivot table to aggregate the rating and idx columns values by sex and decade. For the rating we'll take the average of the reviews for each decade/sex while, for idx we'll get the len (just to count the number of reviews):

In [256]:
```
prefered_decade_by_sex = final_lens.pivot_table(
    columns=['sex', ],
    index=[ 'decade'],
    values=['rating', 'idx'],
    aggfunc={'rating': np.average, 'idx': len},
    fill_value=0,
)
print prefered_decade_by_sex
# We see that there are a bunch of movies without decade, we'll drop th
# Notice that I pass '' to drop to remove the column with empty index
prefered_decade_by_sex = prefered_decade_by_sex.drop('')
prefered_decade_by_sex
```

```
          rating              idx
sex            F         M      F      M
decade
        3.500000  3.428571      2      7
20's    2.857143  3.632653      7     49
30's    3.961340  3.912455    388   1108
40's    3.952641  4.029412    549   1700
50's    3.835006  3.972403    897   2609
60's    3.852321  3.891015    948   2927
70's    3.754007  3.899522   1435   4807
80's    3.700214  3.764700   2802   9320
90's    3.437366  3.384938  18712  51733
```

Out[256]:

| | rating | | idx | |
|---|---|---|---|---|
| **sex** | **F** | **M** | **F** | **M** |
| **decade** | | | | |
| **20’s** | 2.857143 | 3.632653 | 7 | 49 |
| **30’s** | 3.961340 | 3.912455 | 388 | 1108 |
| **40’s** | 3.952641 | 4.029412 | 549 | 1700 |
| **50’s** | 3.835006 | 3.972403 | 897 | 2609 |
| **60’s** | 3.852321 | 3.891015 | 948 | 2927 |
| **70’s** | 3.754007 | 3.899522 | 1435 | 4807 |
| **80’s** | 3.700214 | 3.764700 | 2802 | 9320 |
| **90’s** | 3.437366 | 3.384938 | 18712 | 51733 |

Now, we see that we have the rating and review count for both men and women. However, I’d also like to get their combined (average rating and total review) values. There are two ways that this can be done: First, we can create *another* dataframe that does not seperate sex:

In [257]:
```
prefered_decade = final_lens.pivot_table(
    index=[ 'decade'],
    values=['rating', 'idx'],
    aggfunc={'rating': np.average, 'idx': len},
    fill_value=0,
)
# Drop non needed index
prefered_decade = prefered_decade.drop('')
prefered_decade
```

Out[257]:

| | idx | rating |
|---|---|---|
| **decade** | | |
| **20's** | 56 | 3.535714 |
| **30's** | 1496 | 3.925134 |
| **40's** | 2249 | 4.010671 |
| **50's** | 3506 | 3.937250 |
| **60's** | 3875 | 3.881548 |
| **70's** | 6242 | 3.866069 |
| **80's** | 12122 | 3.749794 |
| **90's** | 70445 | 3.398864 |

Now, these two DataFrames can be easily combined because they have the same index. I'll put the values from the total DataFrame as a second level index to the seperated (by sex) DataFrame - notice how the multi index is used for indexing:

In [258]:
```
prefered_decade_by_sex['rating', 'total'] = prefered_decade['rating']
prefered_decade_by_sex['idx', 'total'] = prefered_decade['idx']
prefered_decade_by_sex
```

Out[258]:

| | rating | | idx | | rating | idx |
|---|---|---|---|---|---|---|
| **sex** | **F** | **M** | **F** | **M** | **total** | **total** |
| **decade** | | | | | | |
| **20's** | 2.857143 | 3.632653 | 7 | 49 | 3.535714 | 56 |
| **30's** | 3.961340 | 3.912455 | 388 | 1108 | 3.925134 | 1496 |
| **40's** | 3.952641 | 4.029412 | 549 | 1700 | 4.010671 | 2249 |
| **50's** | 3.835006 | 3.972403 | 897 | 2609 | 3.937250 | 3506 |
| **60's** | 3.852321 | 3.891015 | 948 | 2927 | 3.881548 | 3875 |
| **70's** | 3.754007 | 3.899522 | 1435 | 4807 | 3.866069 | 6242 |
| **80's** | 3.700214 | 3.764700 | 2802 | 9320 | 3.749794 | 12122 |
| **90's** | 3.437366 | 3.384938 | 18712 | 51733 | 3.398864 | 70445 |

Now, the previous would be almost perfect but I would really prefer the rating and idx first-level columns to be all together. This can be done by using the `sort_index` — the `axis=1` parameter sorts the columns(or else the index will be sorted):

In [259]:
```
prefered_decade_by_sex = prefered_decade_by_sex.sort_index(axis=1)
prefered_decade_by_sex
```

Out[259]:

| | idx | | | rating | | |
|---|---|---|---|---|---|---|
| **sex** | **F** | **M** | **total** | **F** | **M** | **total** |
| **decade** | | | | | | |
| **20's** | 7 | 49 | 56 | 2.857143 | 3.632653 | 3.535714 |
| **30's** | 388 | 1108 | 1496 | 3.961340 | 3.912455 | 3.925134 |
| **40's** | 549 | 1700 | 2249 | 3.952641 | 4.029412 | 4.010671 |
| **50's** | 897 | 2609 | 3506 | 3.835006 | 3.972403 | 3.937250 |
| **60's** | 948 | 2927 | 3875 | 3.852321 | 3.891015 | 3.881548 |
| **70's** | 1435 | 4807 | 6242 | 3.754007 | 3.899522 | 3.866069 |
| **80's** | 2802 | 9320 | 12122 | 3.700214 | 3.764700 | 3.749794 |
| **90's** | 18712 | 51733 | 70445 | 3.437366 | 3.384938 | 3.398864 |

The other method to create the sex and total reviews DataFrame is to aggregate the values from `prefered_decade_by_sex` directly (without creating another pivot table). Take a look at the `get_average` function below. For each row it will take the total number of reviews for men and women and multiply that number with the corresponding average. It will then divide the sum of averages with the total number of reviws to get the average for each row. We also use the `sort_index` method to display the columns correctly:

In [260]:
```
prefered_decade_by_sex = final_lens.pivot_table(
    columns=['sex', ],
    index=[ 'decade'],
    values=['rating', 'idx'],
    aggfunc={'rating': np.average, 'idx': len},
    fill_value=0,
)

prefered_decade_by_sex = prefered_decade_by_sex.drop('')

def get_average(row):
    total_f = row['idx']['F']
    total_m = row['idx']['M']
    total_rating_f = total_f*row['rating']['F']
    total_rating_m = total_m*row['rating']['M']

    return (total_rating_f+total_rating_m)/(total_f+total_m)

prefered_decade_by_sex['rating', 'total'] = prefered_decade_by_sex.appl
prefered_decade_by_sex['idx', 'total'] = prefered_decade_by_sex.apply(l
prefered_decade_by_sex = prefered_decade_by_sex.sort_index(axis=1)
prefered_decade_by_sex
```

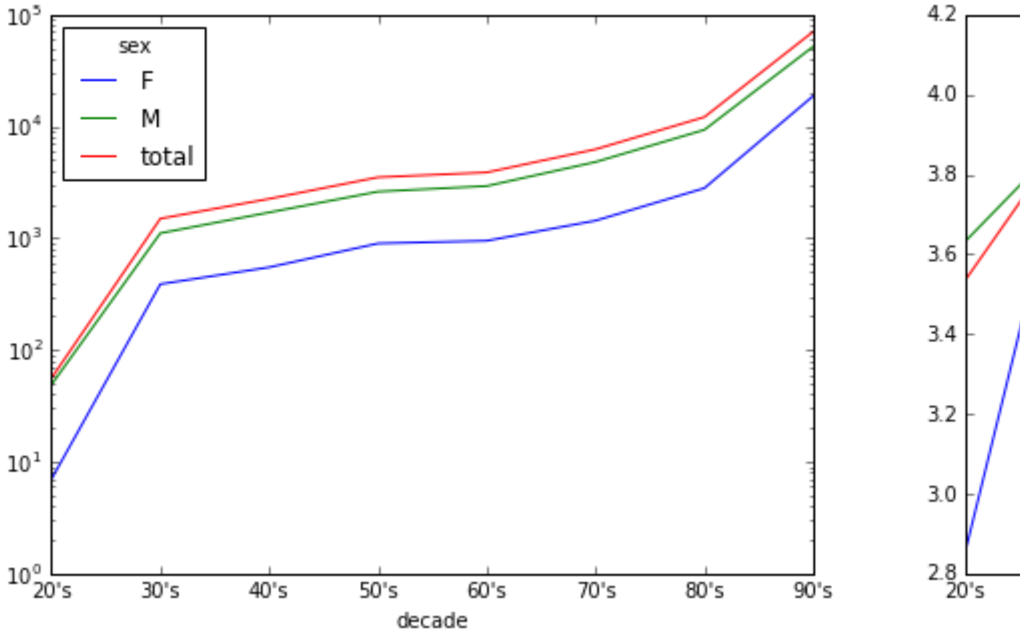
Out[260]:

| | **idx** | | | **rating** | | |
|---|---|---|---|---|---|---|
| **sex** | **F** | **M** | **total** | **F** | **M** | **total** |
| **decade** | | | | | | |
| **20's** | 7 | 49 | 56.0 | 2.857143 | 3.632653 | 3.535714 |
| **30's** | 388 | 1108 | 1496.0 | 3.961340 | 3.912455 | 3.925134 |
| **40's** | 549 | 1700 | 2249.0 | 3.952641 | 4.029412 | 4.010671 |
| **50's** | 897 | 2609 | 3506.0 | 3.835006 | 3.972403 | 3.937250 |
| **60's** | 948 | 2927 | 3875.0 | 3.852321 | 3.891015 | 3.881548 |
| **70's** | 1435 | 4807 | 6242.0 | 3.754007 | 3.899522 | 3.866069 |
| **80's** | 2802 | 9320 | 12122.0 | 3.700214 | 3.764700 | 3.749794 |
| **90's** | 18712 | 51733 | 70445.0 | 3.437366 | 3.384938 | 3.398864 |

Let's try to plot this DataFrame to see if we can extract some useful conclusions:

In [261]:
```
f, a = plt.subplots(1,2)
prefered_decade_by_sex['idx'].plot(ax=a[0], figsize=(15,5), logy=True)
prefered_decade_by_sex['rating'].plot(ax=a[1], figsize=(15,5))
# It seems that women don't like movies from the 20's (but if you take
# of votes there are too few women that have voted for 20's movies. Als
# reviews seem to be for movies of 30's and 40's and (as expected) the
# are for newest movies
```

Out[261]:

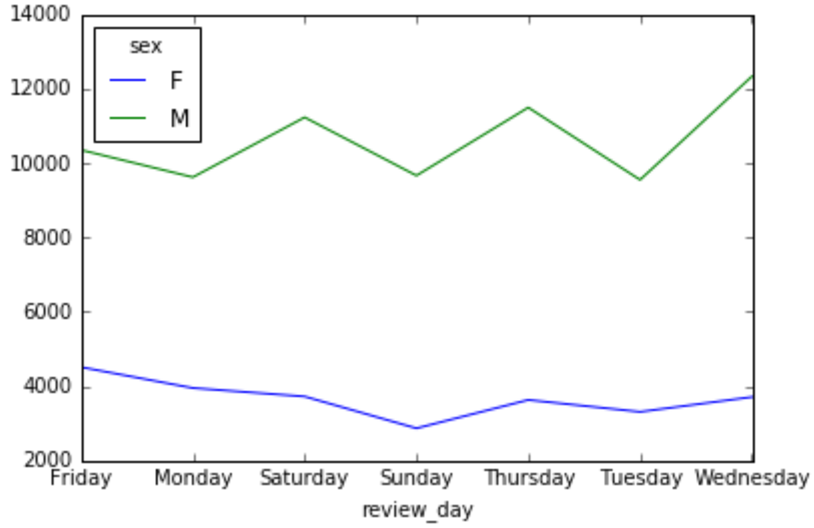



Let's try another quick `pivot_table`. Can we see if there's a specific day-of-week at which the reviewers prefer to vote?

In [262]:

```
final_lens.pivot_table(
    index=['review_day'],
    columns=['sex'],
    values='idx',
    aggfunc=len
).plot()

# Probably not
```

Out[262]:




Continuing our exploration of the movie-reviews data set, we'd like to get the total number of reviews and average rating for each movie. To make a better display for the release we'll categorise the movies by their decade and release year (using multi index rows):

In [263]:
```
rating_count = final_lens.pivot_table(
    index=[ 'decade', 'release_year', 'title',],
    values=['rating', 'idx', ],
    aggfunc={
        'rating': np.average,
        'idx': len,

    }
)
# Drop movies without decade
rating_count = rating_count.drop('')
rating_count.head(10)
# Notice the nice hierarchical index on decade and release_year
```

Out[263]:

| | | | idx | rating |
|---|---|---|---|---|
| **decade** | **release_year** | **title** | | |
| **20's** | **1922** | **Nosferatu (Nosferatu, eine Symphonie des Grauens) (1922)** | 54 | 3.555556 |
| | **1926** | **Scarlet Letter, The (1926)** | 2 | 3.000000 |
| **30's** | **1930** | **Blue Angel, The (Blaue Engel, Der) (1930)** | 18 | 3.777778 |
| | **1931** | **M (1931)** | 44 | 4.000000 |
| | **1932** | **Farewell to Arms, A (1932)** | 12 | 3.833333 |
| | **1933** | **Duck Soup (1933)** | 93 | 4.000000 |
| | | **Liebelei (1933)** | 1 | 1.000000 |
| | **1934** | **Gay Divorcee, The (1934)** | 15 | 3.866667 |
| | | **It Happened One Night (1934)** | 81 | 4.012346 |
| | | **Of Human Bondage (1934)** | 5 | 3.200000 |

As we can see above, there are movies with very few revies. I don't really want to count them since they'll probably won't have correct ratings. Also, instead of displaying all the movies I'd like to create a small list with only the best movies:

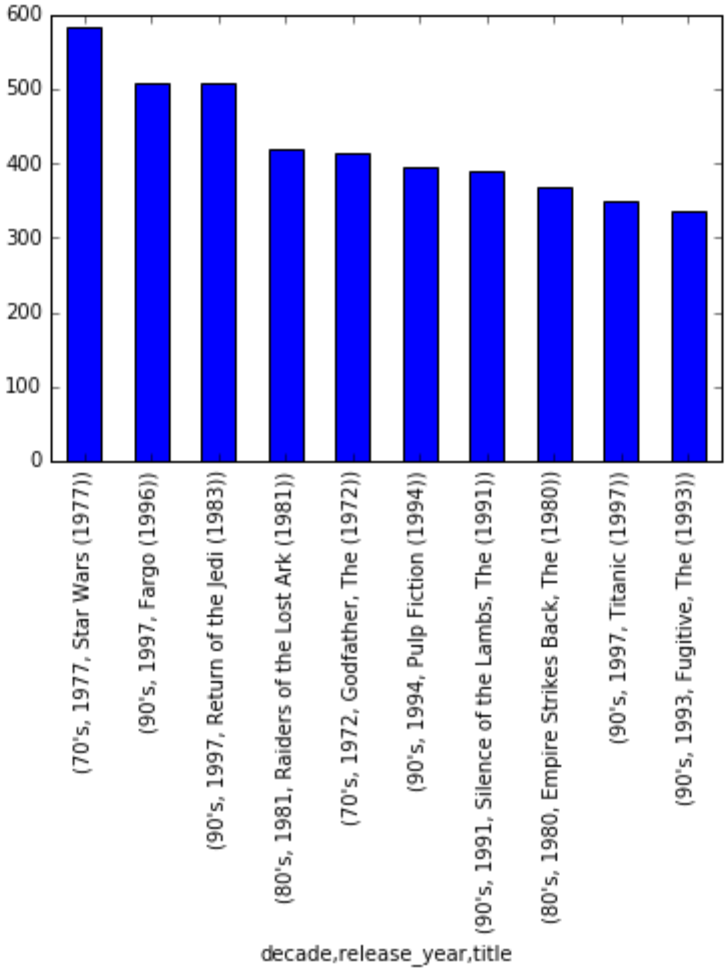
In [264]:
```
# So, let's find the best movies (rating more than 4) with more than 15
best_movies = rating_count[(rating_count.idx>150) & (rating_count.ratin
best_movies
```

Out[264]:

| | | | idx | rating |
|---|---|---|---|---|
| **decade** | **release_year** | **title** | | |
| **30's** | **1939** | **Wizard of Oz, The (1939)** | 246 | 4.077236 |
| **40's** | **1941** | **Citizen Kane (1941)** | 198 | 4.292929 |
| | **1942** | **Casablanca (1942)** | 243 | 4.456790 |
| | **1946** | **It's a Wonderful Life (1946)** | 231 | 4.121212 |
| **50's** | **1951** | **African Queen, The (1951)** | 152 | 4.184211 |
| | **1954** | **Rear Window (1954)** | 209 | 4.387560 |
| | **1957** | **Bridge on the River Kwai, The (1957)** | 165 | 4.175758 |
| | **1958** | **Vertigo (1958)** | 179 | 4.251397 |
| | **1959** | **North by Northwest (1959)** | 179 | 4.284916 |
| **60's** | **1960** | **Psycho (1960)** | 239 | 4.100418 |
| | **1962** | **Lawrence of Arabia (1962)** | 173 | 4.231214 |
| | | **To Kill a Mockingbird (1962)** | 219 | 4.292237 |
| | **1963** | **Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1963)** | 194 | 4.252577 |
| | **1967** | **Graduate, The (1967)** | 239 | 4.104603 |
| **70's** | **1972** | **Godfather, The (1972)** | 413 | 4.283293 |
| | **1973** | **Sting, The (1973)** | 241 | 4.058091 |
| | **1974** | **Godfather: Part II, The (1974)** | 209 | 4.186603 |
| | | **Monty Python and the Holy Grail (1974)** | 316 | 4.066456 |
| | **1975** | **One Flew Over the Cuckoo's Nest (1975)** | 264 | 4.291667 |
| | **1977** | **Star Wars (1977)** | 583 | 4.358491 |
| | **1979** | **Alien (1979)** | 291 | 4.034364 |
| | | **Apocalypse Now (1979)** | 221 | 4.045249 |
| **80's** | **1980** | **Empire Strikes Back, The (1980)** | 367 | 4.204360 |
| | **1981** | **Raiders of the Lost Ark (1981)** | 420 | 4.252381 |
| | **1982** | **Blade Runner (1982)** | 275 | 4.138182 |
| | | **Gandhi (1982)** | 195 | 4.020513 |
| | **1984** | **Amadeus (1984)** | 276 | 4.163043 |
| | **1987** | **Princess Bride, The (1987)** | 324 | 4.172840 |
| | **1989** | **Glory (1989)** | 171 | 4.076023 |
| **90's** | **1991** | **Silence of the Lambs, The (1991)** | 390 | 4.289744 |
| | | **Terminator 2: Judgment Day (1991)** | 295 | 4.006780 |
| | **1993** | **Fugitive, The (1993)** | 336 | 4.044643 |
| | | **Much Ado About Nothing (1993)** | 176 | 4.062500 |
| | | **Schindler's List (1993)** | 298 | 4.466443 |
| | **1994** | **Pulp Fiction (1994)** | 394 | 4.060914 |
| | | **Shawshank Redemption, The (1994)** | 283 | 4.445230 |
| | **1995** | **Sense and Sensibility (1995)** | 268 | 4.011194 |

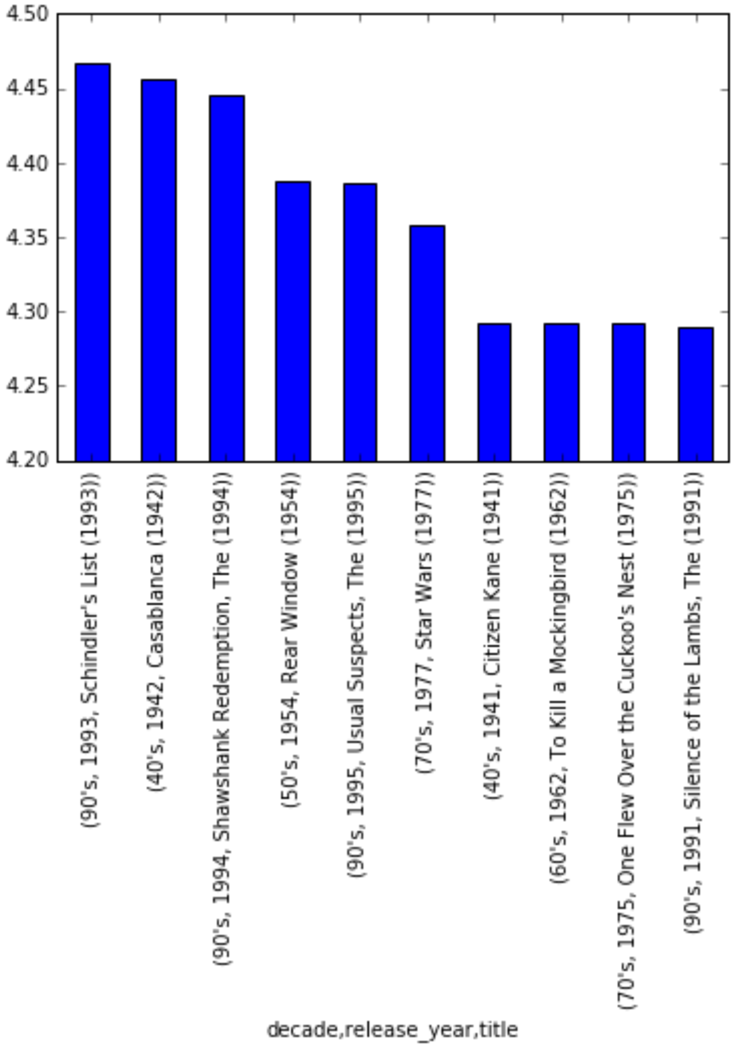
In [265]:
```
# Which are the most popular movies (number of votes) ?
best_movies.sort_values(by='idx', ascending=False)['idx'][:10].plot(kin

# Fargo at the 2nd and Fugitive at the 10nth place of popularity seem a
```

Out[265]:



In [266]:
```
# Which are the best movies (vote average) ?
best_movies.sort_values(by='rating', ascending=False)['rating'][:10].pl

# I tend to agree with most of them, however I feel that the Godfather
```

Out[266]:




Let’s try to see how many people of each age are voting, however instead of displaying the votes for people of each specific age, we’ll seperate the ages into groups (0-10, 10,20 etc) and dispaly the counts for them:

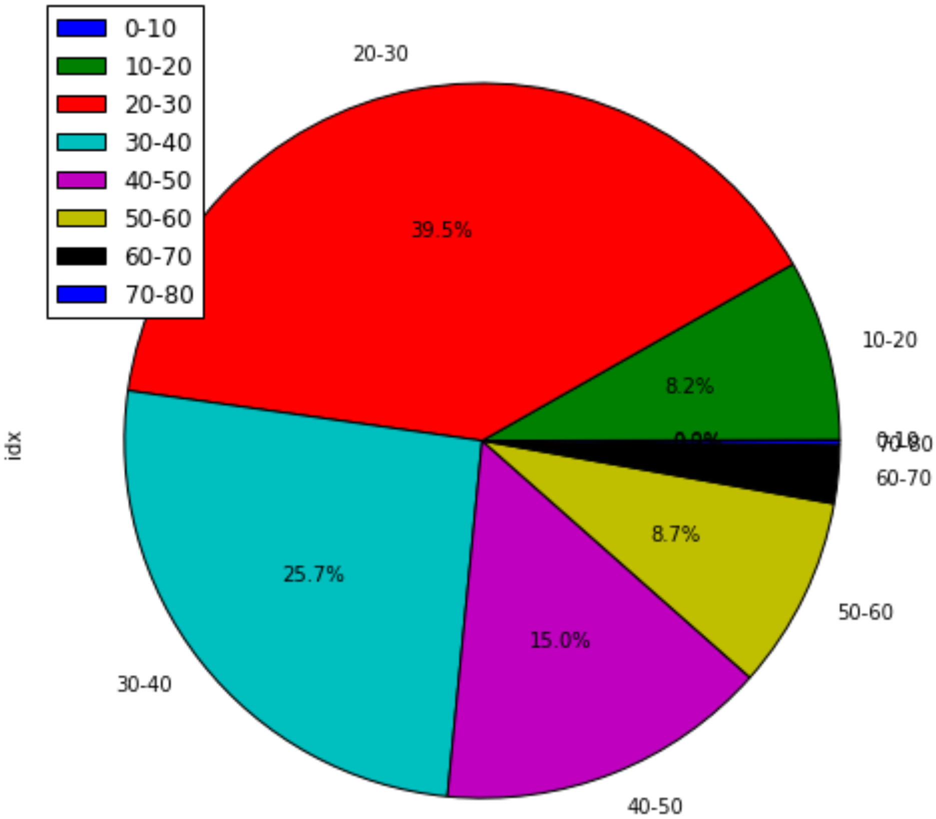
In [267]:
```
review_idx_by_age = final_lens.pivot_table(index=['age'], values='idx',
print review_idx_by_age.head(10)

# Let's group by age group
def by_age(x):
    return '{0}-{1}'.format((x/10)*10, (x/10 + 1)*10)


grouped_review_idx = review_idx_by_age.groupby(by_age).aggregate(sum)
grouped_review_idx
```

```
age
7       43
10      31
11      27
13     497
14     264
15     397
16     335
17     897
18    2219
19    3514
Name: idx, dtype: int64
```

Out[267]:
```
0-10        43
10-20     8181
20-30    39535
30-40    25696
40-50    15021
50-60     8704
60-70     2623
70-80      197
Name: idx, dtype: int64
```

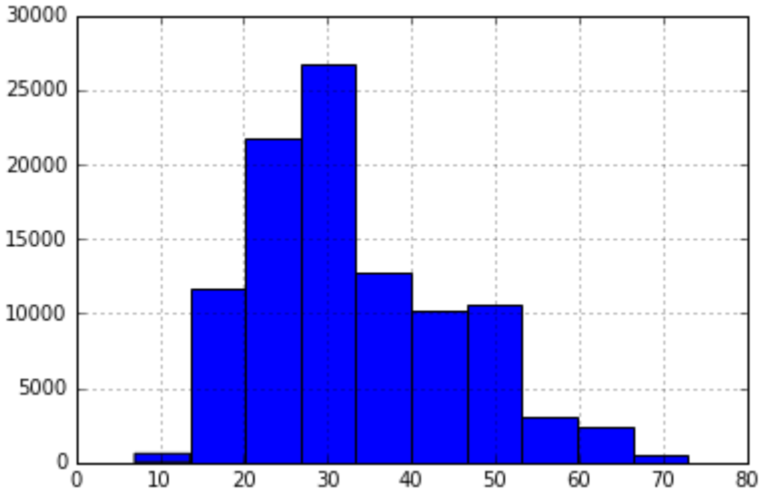
In [268]:
```
# Let's plot our number of votes - we can see that most people voting a
grouped_review_idx.plot(kind='pie', figsize=(8, 8), legend=True, autopc
```

Out[268]:



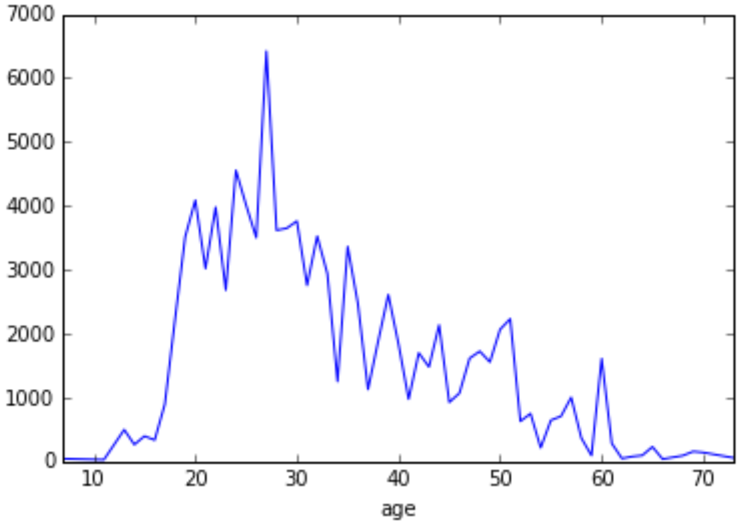

In [269]:
```
# We can also see the same (more or less) info directly from the
# initial dataset, using a histogram
final_lens['age'].hist(bins=10)
```

Out[269]:



In [270]:
```
# Or just by plotting the num of reviews / age (not the grouped one)
review_idx_by_age.plot()
```

Out[270]:




Let’s see how people of each occupation and sex are voting. We’ll get the averages for age, and rating and total number of reviews:

In [271]:

```
aggrs = {'age': np.average, 'idx': len, 'rating' :np.average}
# This creeates a dataframe for men and women
d1 = final_lens.pivot_table(index='occupation', columns='sex', aggfunc=
# This creates a dataframe for both
d2 = final_lens.pivot_table(index='occupation', aggfunc=aggrs)
# Let's put the values from the "both" dataframe to the men/women dataf
d1[ 'idx','Total' ] = d2['idx']
d1[ 'age','Total'] = d2['age']
d1[ 'rating','Total'] = d2['rating']
# And now let's sort the row index so that it will have the correct mul
occupations = d1.sort_index(axis=1)
# Finally, let's sort the DataFrame by the total number of votes for ea
occupations = occupations.sort_values( ('idx', 'Total'), ascending=Fals
occupations
# Students are no1 - not a surpise!
```

Out[271]:

| | age | | | idx | | | |
|---|---|---|---|---|---|---|---|
| **sex** | **F** | **M** | **Total** | **F** | **M** | **Total** | **F** |
| **occupation** | | | | | | | |
| **student** | 21.092697 | 22.510731 | 22.142870 | 5696.0 | 16261.0 | 21957 | 3.6028 |
| **other** | 31.813370 | 32.964704 | 32.568977 | 3665.0 | 6998.0 | 10663 | 3.5312 |
| **educator** | 37.942058 | 44.570167 | 42.789240 | 2537.0 | 6905.0 | 9442 | 3.6988 |
| **engineer** | 33.489655 | 34.371731 | 34.356086 | 145.0 | 8030.0 | 8175 | 3.7517 |
| **programmer** | 32.463007 | 32.502167 | 32.500064 | 419.0 | 7382.0 | 7801 | 3.5775 |
| **administrator** | 38.096081 | 39.688083 | 39.123145 | 2654.0 | 4825.0 | 7479 | 3.7818 |
| **writer** | 37.429848 | 32.219527 | 34.325867 | 2238.0 | 3298.0 | 5536 | 3.6639 |
| **librarian** | 36.707343 | 38.129300 | 37.358050 | 2860.0 | 2413.0 | 5273 | 3.5800 |
| **technician** | 38.000000 | 31.512655 | 31.712493 | 108.0 | 3398.0 | 3506 | 3.2685 |
| **executive** | 42.529412 | 36.203331 | 36.614164 | 221.0 | 3182.0 | 3403 | 3.7737 |
| **healthcare** | 37.644993 | 44.641851 | 38.885164 | 2307.0 | 497.0 | 2804 | 2.7360 |
| **artist** | 27.155510 | 33.088257 | 30.592288 | 971.0 | 1337.0 | 2308 | 3.3470 |
| **entertainment** | 27.546667 | 28.912834 | 28.766110 | 225.0 | 1870.0 | 2095 | 3.4488 |
| **scientist** | 28.273381 | 35.855133 | 35.343052 | 139.0 | 1919.0 | 2058 | 3.2517 |
| **marketing** | 32.106335 | 37.759947 | 36.478462 | 442.0 | 1508.0 | 1950 | 3.5226 |
| **retired** | 70.000000 | 61.375163 | 61.755749 | 71.0 | 1538.0 | 1609 | 3.2394 |
| **lawyer** | 36.000000 | 34.478056 | 34.556134 | 69.0 | 1276.0 | 1345 | 3.6231 |
| **none** | 33.887671 | 18.960821 | 25.007769 | 365.0 | 536.0 | 901 | 3.6328 |
| **salesman** | 31.318584 | 34.882012 | 33.470794 | 339.0 | 517.0 | 856 | 3.8702 |
| **doctor** | NaN | 35.592593 | 35.592593 | NaN | 540.0 | 540 | NaN |
| **homemaker** | 33.416357 | 23.000000 | 32.371237 | 269.0 | 30.0 | 299 | 3.2788 |

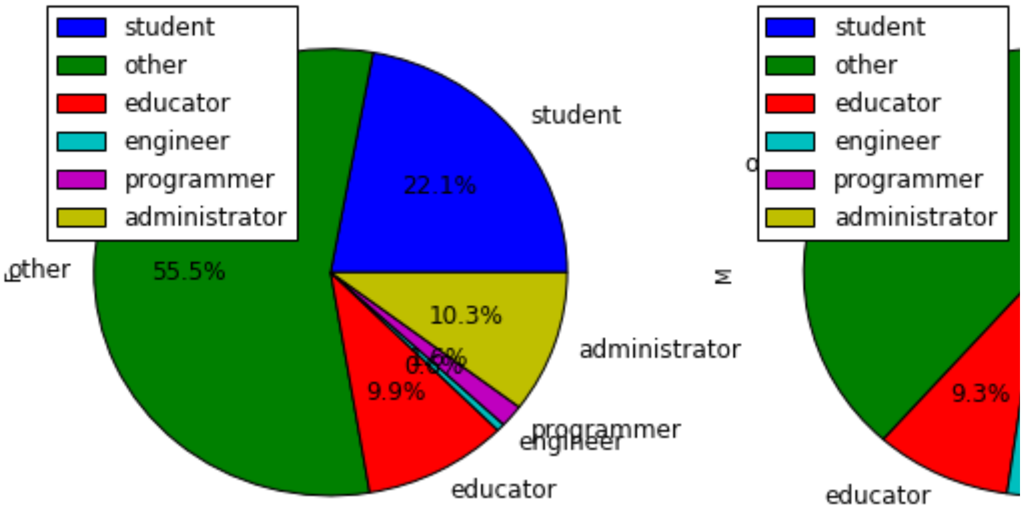
Let's manipulate the previous DataFrame to take a look at only the occupations that have voted the most number of times. Actually, we'll get only occupations that have voted more than 6000 times, all other occupations we'll just add them to the "other" occupation. For this, we'll get only the `'idx'` column and filter it by the rows which have a `Total<6000`. We'll then take the sum for this DataFrame so that we'll get the total votes for each male/female and total.

Next, we'll add this to the "other" row of the dataframe and remove the less than 6000 rows from it. Finally, we'll plot the resulting DataFrame for all male, female and both.

In [272]:
```
occupations_num = occupations['idx']
# Let's see which are the total numbers we need to add to "other"
add_to_other = occupations_num[occupations_num['Total']<6000].sum()
print add_to_other

occupations_num.loc['other']+=add_to_other
print occupations_num.loc['other']

# Now let's get the rows that have a total of > 6000
most_voters_with_other = occupations_num[occupations_num['Total']>6000]
print most_voters_with_other
most_voters_with_other.plot(kind='pie', subplots=True, figsize=(18,5 ),
```

```
sex
F        10624.0
M        23859.0
Total    34483.0
dtype: float64
sex
F        14289.0
M        30857.0
Total    45146.0
Name: other, dtype: float64
sex                  F        M    Total
occupation
student         5696.0  16261.0  21957.0
other          14289.0  30857.0  45146.0
educator        2537.0   6905.0   9442.0
engineer         145.0   8030.0   8175.0
programmer       419.0   7382.0   7801.0
administrator   2654.0   4825.0   7479.0
```

Out[272]:
```
array([,
       ,
       ], dtype=object)
```



## Conclusion

In the previous, I have tried to present a comprehensive introduction of the `pivot_table` command along with a small introduction to the pandas data structures (`Series` and `DataFrame`) and a bunch of other methods that will help in using `pivot_table` presenting both toy and real world examples for all of them. If you feel that something is missing or you want me to add another example pivot_table operation for either the books or the movie lens dataset feel free to tell me in the comments.

I hope that after reading (and understanding) the above you'll be able to use pandas and `pivot_table` without problems!

Posted by Serafeim Papastefanos Τετ 21 Σεπτέμβριος 2016 python python, pandas, scipy, numpy, pivot, pivot_table, ipython, jupyter, notebook

Tweet

## Comments

0 Comments spapas.github.io

1 Login

Recommend 1 Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON SPAPAS.GITHUB.IO

**Understanding nested list comprehension syntax in Python**

4 comments • a year ago•

pras0784 — Awesome description of Python nested list comprehension.

**django-rq redux: advanced techniques and tools**

5 comments • 2 years ago•

Itamar Haber — But of course - here's a little something I've written on subject:https://redislabs.com/blog/...

**PDFs in Django: The essential guide**

12 comments • 2 years ago•

spapas — Hello! The django-xhtml2pdf is actually a thin wrapper around xhtml2pdf, which does all the work. What django-

**A comprehensive React and Flux tutorial part 1: React**

7 comments • 2 years ago•

Raul Piaggio — Hi Serafeim, thank you for this! Quite enlightening example and explanation. I'm actually delving into React.js

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

Atom | RSS

### Recent Posts

- Automatically create a category table in Postgresql by extracting unique table values
- Creating custom components for ag-grid
- Getting a logical backup of all databases of your Postgresql server
- A pandas pivot_table primer
- Splitting a query into individual fields in Django

### Categories

- css
- django
- flask

- git
- javascript
- pelican
- postgresql
- python
- spring
- wagtail

### Tags

less, rest, query, tables, pivot_table, console, django-rq, github-pages, forms, yaml, 404, ldap, spring-security, init.d, python, django-extensions, class-based-views, scipy, grid, ag-grid, watchify, settings, flux, FixedDataTable, debug, imgur, security, node, google, spring, react-redux, cron, babelify, design, tutorial, gmail, reportlab, rq, redis, research, werkzeug, jobs, redux-thunk, javascript, component, ajax, ipython, java, configuration, properties, bash, es6, spring-boot, postgresql, filter, flask, static-html, pdf, react-notification, pelican, tasks, node.js, pivot, auditing, git, uglify, introduction, pusher, fixed-data-tables, heroku, xhtml2pdf, numpy, boilerplate, wagtail, rst, profiles, github.io, error, cbv, react-router-redux, history, django-rest-framework, redux-form, asynchronous, scheduling, plpgsql, generic, babel, authentication, pandas, css, npm, deploy, mongodb, react, notebook, browserify, branching, github, jupyter, windows, django, q, react-router, redux, boostrap-material-design

Copyright © 2013–2017 Serafeim Papastefanos — Powered by Pelican