Read time: 21 minutes

Pandas & Seaborn - A guide to handle & visualize data elegantly

Eugenio Thu, Mar 16, 2017 in **#MACHINE LEARNING**

MACHINE LEARNING BACKEND

Here at Tryolabs we love Python almost as much as we love machine learning problems. These kind of problems always involve working with large amounts of data which is key to understand before applying any machine learning technique. To understand the data, we need to manipulate it, clean it, make calculations and see how variables behave independently, and how they relate to one another. At this post will show how we have been doing this lately.

Up next you'll find an overview of **Pandas**, a Python library which is old but gold and a must-know if you're attempting to do any work with data while living in the Python world, and a glance of **Seaborn**, a Python library for making statistical visualizations. From our experience, they complement each other really well, and are worth learning together. We hope this post serves as a first guide for diving into them and kickstart your data handling & visualization journey.

Pandas

Why is Pandas great? It is built on top on **NumPy**. It uses its multi-dimensional arrays and fast operations internally to provide higher level methods for manipulation and analysis.

It is also easy to use. Almost every Pandas method returns a (modified) copy of the data, which allows you to chain transformations, and perform complex modifications in one line. The overview is divided into sections, each one with code examples and an explanation of what is being done.

First we'll need to make some imports, which will be necessary through all the examples. We'll use the well known Titanic dataset (available in Seaborn), which holds data of the Titanic passengers, such as their age, paid fare, and if they survived or not.

```python
import numpy as np
import pandas as pd
import seaborn as sns
import timeit

# Load dataset
titanic = sns.load_dataset('titanic')
```

view raw

Basic aspects

These are must knows that will make your life easier when dealing with Pandas for the first time.

Data Structures

Pandas' data structures can hold mixed typed values as well as labels, and their axes can have names set. The data structures are the following.

The most basic Data Structure available in Pandas is the **Series**. This is basically a 1-dimensional labeled array. Therefore, Series have only one axis (axis == 0) called "index".

```python
pd.Series([1, 90, 'hey', np.nan], index=['a', 'B', 'C', 'd'])
```

view raw

a	B	C	d
1	90	"hey"	NaN

Then, we have **DataFrames**. These are 2-dimensional structures, with two axes, the "index" axis (axis == 0), and the "columns" axis (axis == 1). DataFrames can be thought of as Python dictionaries where the keys are the column labels, and the values are the column Series.

```python
pd.DataFrame({'day': [17, 30], 'month': [1, 12], 'year': [2010, 2017]})
```

view raw

	day	month	year

0	17	1	2010
1	30	12	2017

Lastly, we have **Panels**. These are 3-dimensional data structures, that are rarely used, in comparison with DataFrames. Analogously to DataFrames, they can be thought of as Python dictionaries of DataFrames. Instead of “index” and “columns”, Panels’ axes are named as follow:

- items (axis == 0)
- major_axis (axis == 1)
- minor_axis (axis == 2)

The axes distinction is vital, since a lot of methods need to have this specified properly in order to work as expected. We’ll see it’s usage in following examples.

From here on, we will use the Series/DataFrame as the in the examples when explaining things, if something applies to a Series it probably does for DataFrames too, but it may not work the other way around.

Getting information and basic calculations

Pandas offers some methods to get information of a data structure: *info, index, columns, axes*, where you can see the memory usage of the data, information about the axes such as the data types involved, and the number of not-null values.

```
titanic.info()
Out[]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 891 entries, 0 to 890
Data columns (total 15 columns):
survived       891 non-null int64
pclass         891 non-null int64
sex            891 non-null object
age            714 non-null float64
sibsp          891 non-null int64
parch          891 non-null int64
fare           891 non-null float64
embarked       889 non-null object
class          891 non-null object
who            891 non-null object
adult_male     891 non-null bool
```

```
deck           203 non-null object
embark_town    889 non-null object
alive          891 non-null object
alone          891 non-null bool
dtypes: bool(2), float64(2), int64(4), object(7)
memory usage: 99.2+ KB
```

view raw

It also offers methods to do basic calculations such as *count, mean, max, min, cumsum, imax, imin*. There's a method called *describe* which is great, because it does almost all of the calculations just mentioned, and presents them in a nicely formatted table.

We also have **sorting** and **ranking** methods. For example, to sort the data based on its index, or on any column we have *sort_index* and *sort_values*. To calculate the ranking of a row based on a value, there's the *rank* method, which is pretty straightforward to use.

In order to perform operations between data structures, we can use **arithmetic and boolean operators**. Amongst them we have +, -, *, /, **, &, |, ~, etc. An alternative to some of these are the *div, add, multiply and subtract* methods, which are equivalent to the corresponding operators, but with fill_value support in case of NaN values.

Accessing and setting data

There are **several methods** to retrieve data from a DataFrame, they can be separated by whether they return a scalar/subset of data, or if they are label/position based.

	Label based	Position based
Returns scalar	*at*	*iat*
Returns subset of data	*loc*	*iloc*

There's a method which is label and posibion based, the *ix* method. It returns a subset of data and is primarily label based, with a fallback to integer positional access, with an exception. If the axis' labels are integer typed, then the method won't fall back to integer positional access. In this case, if you want to use positions, you need to use the *iat* or *iloc* methods.

```python
# Returns a scalar
titanic.ix[4, 'age']

# Returns a Series of name 'age', and the age values associated to the index labels 4 and 5
titanic.ix[[4, 5], 'age']

# Returns a Series of name '4', and the age and fare values associated to that row.
titanic.ix[4, ['age', 'fare']]

# Returns a DataFrame with rows 4 and 5, and columns 'age' and 'fare'
titanic.ix[[4, 5], ['age', 'fare']]
```

view raw

Positional access allows you to slice a DataFrame, just like you do with lists in Python. Slice notation works in the *iat, iloc*, and *ix* methods.

These methods can also be used to set data, from scalars to a set of rows.

Boolean Queries

If you want to get a subset of the data but you don't know the associated labels/positions, a boolean query is what you want.

To make a boolean query, you need to pass the DataFrame a True/False Series whose index aligns with the DataFrame being queried's index. That is, for every label in the DataFrame, there must be True/False value associated to that same label in the Series.

The True/False values in the Series indicate if the value associated to a certain label should be returned in the query or not.

```python
A = pd.Series(['a', 'b', 'c', 'd', 'e', 'f'])
A_filter = pd.Series([False, True, False, True, False, True, True])
A[A_filter]
Out[]:
1    b
3    d
5    f
dtype: object
```

view raw

In this example, we run a boolean query on a Series, instead of a DataFrame, and of course, it worked as expected.

The Series can be assembled combining other boolean queries (the query “terms”) using boolean operators.

Example query

```
titanic[
    (titanic.sex == 'female')
    & (titanic['class'].isin(['First', 'Third']))
    & (titanic.age > 30)
    & (titanic.survived == 0)
]
```

view raw

- The query has four True/False series (query terms) combined with the “&” (element to element) operator.
- The first Series was constructed using the titanic.sex Series ([‘female’, ‘male’, ‘female’…]) and comparing each value to the ‘female’ string, returning True/False for each value.
- The second Series used the *pandas.Series.isin* function (which is pretty straightforward), and also accesses the series using square brackets for a reason; “class” is a reserved word in Python for defining classes, and using it with the dot notation raises a SyntaxError exception.
- The third and fourth “terms” work just like the first one.

Some cool features

Indexing

One of the benefits of Pandas data structures being in-memory is that it allows fast indexing. You can set, and remove a DataFrame’s index with the methods *set_index,* and *reset_index* respectively.

There’s a method called *reindex* that allows you to conform a DataFrame to a new index both for rows and columns, with an optional filling logic (by default, new row/column labels will have NaN values associated in the new DataFrame). If the original DataFrame has duplicated labels in its row/column index then reindex will fail. You will need to either reset the index prior to the reindexing, or remove rows/columns with duplicated labels.

Alignment

One property of data stored in Pandas' data structures is that it is aligned to the structure's axes. This means that, when performing an operation, between two data structures, what matters is the associated label, and not the order of the data.

Example

```
s1 = pd.Series([10, 20, 30, np.nan], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([100, 200, 300, 400], index=['e', 'd', 'c', 'b'])
s1.add(s2)
Out[]:
a      NaN
b    420.0
c    330.0
d      NaN
e      NaN
dtype: float64
```

view raw

As we can see, the method took values whose labels matched in order to perform the sum in spite of them being in different positions in the original Series. The result Series' index is aligned with both the series s1 and s2.

Pandas offers a way to perform aligning of data structures artificially, the *align* method.

Calling the align method with the two series from the previous example will result in a tuple of two new Series, the series s1 with its index extended containing the values in s2's index, and the series s2 extended in the same way.

```
s1.align(s2)
Out[]:
(a    10.0
 b    20.0
 c    30.0
 d     NaN
 e     NaN
 dtype: float64, a      NaN
 b    400.0
 c    300.0
 d    200.0
 e    100.0
 dtype: float64)
```

view raw

When aligning structures of higher dimensions, you can specify the axis along which the alignment will be made. For example, when aligning two DataFrames you can align along the 'rows' (axis=0) axis, the 'columns' (axis=1) axis or both (default behaviour).

Combining Data Frames

Pandas offers several methods to combine DataFrames, that can be separated into two approaches, which are **concatenation** and **merging**.

Concatenate DataFrames basically means "sticking them together" along an axis, regardless of the values contained within them. Pandas' main method for concatenation is *concat*, although you can use *append* too!

concat receives a list of DataFrames and uses axis = 0 ('rows') by default, that means that it sticks one DataFrame "below" the ones before. If axis = 1 ('columns') is passed, then it will stick each DataFrame at the right side of the ones before. *concat*, as opposed to several other methods, is only available in the main **pandas namespace**.

A namespace basically is "where a method or attribute" lives. This means, that e.g. concat, can not be invoked from a DataFrame object (it does not live in the pandas.DataFrame namespace).

```python
df1 = pd.DataFrame({'a': [1]})
df2 = pd.DataFrame({'b': [2]})

# 1) Incorrect: Raises AttributeError since concat is not in the DataFrame namespace
df1.concat(df2)

# 2) Correct
pd.concat([df1, df2])

# 3) Correct, and equivalent to 2)
df1.append(df2)

# 4) Incorrect: append does not accept the 'axis' arg. Should use pd.concat with axis='column
df2.append(df2, axis='columns')
```

view raw

Merging is another way of combining DataFrames, but unlike *concat* it combines them looking for matching values in columns of said DataFrames (you can merge by index too). The main method to perform merging in Pandas is *merge* which lives both in the main pandas namespace and in the DataFrame namespace (unlike *concat*).

Its behaviour as is is just like an INNER JOIN in SQL

Example: To show how to merge two DataFrames and some convenient arguments the *merge* function has, I'll create a towns_df DataFrame to merge with the titanic dataset DataFrame.

```python
# Generate a small DataFrame of cities with their population,
# and fake ages (years with the city status)
towns_dic = {
    'name': ['Southampton', 'Cherbourg', 'Queenstown', 'Montevideo'],
    'country': ['United Kingdom', 'France', 'United Kingdom', 'Uruguay'],
    'population': [236900, 37121, 12347, 1305000],
    'age': [np.random.randint(500, 1000) for _ in range(4)]
}
towns_df = pd.DataFrame(towns_dic)
```

view raw

	age	country	name	population
0	570	United Kingdom	Southampton	236900
1	706	France	Cherbourg	37121
2	840	United Kingdom	Queenstown	12347
3	645	Uruguay	Montevideo	1305000

Merge example

```python
(titanic.merge(
  towns_df,
  left_on='embark_town', right_on='name',
  how='left',
  indicator=True,
  suffixes=('_passenger', '_city')
)).head()
# 'head' takes the last n elements of the DataFrame
```

view raw

- **left_on** and **right_on**: We want to combine both DataFrames' rows when there's a match between the "embark_town" column in the titanic DataFrame, and the

"name" column in the towns_df DataFrame, so we indicate this with the **left_on** and **right_on** arguments.
- **how**: By default, *merge* performs an inner join, but we don't want to lose the rows in the original DataFrame that didn't have an associated embark_town value, so we indicate that we want a "left" join with the **how** parameter.
- **indicator**: We also want to know, for each row, if there was a match on both DataFrames, or if the resulting row was only on either one. The **indicator** parameter adds a "_merge" column that indicates this.
- **suffixes**: Since we have "age" columns in both DataFrames, we want to be able to distinguish them properly in the resulting merged DataFrame, we add proper **suffixes** for this purpose, and get "age_passenger" and "age_city" columns in the merged DataFrame as a result.

GroupBy

Pandas' GroupBy is exactly what you'd expect and much more. It is not just a *groupby* method that works like SQL's "GROUP BY" but a whole set of methods to perform splitting into groups, transforming them (perhaps independently) and combining the results. You should definitely check out the **Group By: split-apply-combine** section in the Pandas docs to really get to know (and appreciate) Pandas' GroupBy capabilities.

The main method for grouping data is *groupby.* It exists in the pandas.DataFrame namespace so you can invoke it directly from a DataFrame object, simply by passing a list of the columns you wish to group the DataFrame by.

You can group by any axis. Of course, by default the grouping is made via the index (rows) axis, but you could group by the columns axis.

The *groupby* method is lazy, that is, it doesn't really perform the data splitting until the group is really needed, which is the most practical/efficient way to go in the majority of cases.

In the example, I'll show a really cool Pandas method called *cut* that will allow us to bin the data according to a column. Then we'll split the titanic dataset into several groups with *groupby*.

```
bins = [0, 12, 17, 60, np.inf]
labels = ['child', 'teenager', 'adult', 'elder']
age_groups = pd.cut(titanic.age, bins, labels=labels)
```

```python
titanic['age_group'] = age_groups
```
view raw

- In the example, we want to classify each titanic passenger by their age. Firstly we define the bins, i.e. the values that will delimit the age intervals we want.
- Secondly, we define the labels, i.e. the name of each bin/interval
- Thirdly, we use *pandas.cut* to map the titanic's passengers ages to a label.
- Finally, we set the result Series from the last step as a new column in our titanic DataFrame

Now, we can use the newly created "age_group" column to group by the titanic passengers.

```python
groups = titanic.groupby(['age_group', 'alone'])
```
view raw

"groups" is an object of type DataFrameGroupBy. It can be iterated in order to inspect or work with the different groups (each one a DataFrame object) You can also use the *get_group* method (in this case passing it a 2-sized tuple) to get the corresponding DataFrame group.

The first thing I'd want to see about the resulting groups is the size of each one. To do this you can use the *size* method, which will return a multi-level Series. Each level corresponds to a grouping field, and the values correspond to the number of rows associated to the group.

```
groups.size()
Out[]:
age_group  alone
child      False     67
           True       2
teenager   False     23
           True      21
adult      False    216
           True     363
elder      False      4
           True      18
dtype: int64
```
view raw

Since this result is a Series, we can easily get the size of each group in relation to the whole dataset.

```
100 * groups.size() / len(titanic)
Out[]:
age_group  alone
child      False     7.519641
           True      0.224467
teenager   False     2.581369
           True      2.356902
adult      False    24.242424
           True     40.740741
elder      False     0.448934
           True      2.020202
dtype: float64
```

view raw

Now we can see very easily that, for example, most children did not travel alone.

Calculated columns

When you're working with Pandas, there is something you most certainly will want to do, and that is adding a column with calculated values to your DataFrame.

There're several ways to do this, in fact we've already done it with *pandas.cut* in the "Group By" section, but that was a particular case.

A more general approach is to use the *apply* function. If we wanted to add a **column** "is_old" with boolean values, for **every row** (passenger) of the titanic, we should do the following

```
def is_old_func(row):
    return row.age
titanic['is_old'] = titanic.apply(is_old_func, axis='columns')
```

view raw

What we did there was to create a Series by applying a custom function to each row of the DataFrame. Then we assigned the series to a new column named "is_old" in the titanic DataFrame.

Notice how we use the axis argument in the *DataFrame.apply* invocation. A good way to remember how to use this argument is the following:

- **axis='columns'** makes the custom function receive a Series with **one value per column** (i.e. a row) in **each invocation**.

- **axis='rows'** makes the custom function receive a Series with **one value per row** (i.e. a column) in **each invocation.**

This approach is good if we need to use multiple values of a row. But in this case, we only use the "age" value of every row. So, in this case it would seem unnecessary to use *apply* for the whole DataFrame.

```
def is_old_func_series(value):
    return value > 60
titanic['is_old'] = titanic.age.apply(is_old_func_series)
```

view raw

- *Series.apply* does not receive an "axis" argument since Series have only one axis.
- We defined a new *is_old_func_series* custom function since Series.apply passes scalars (the Series values) to the custom function, and trying to access the attribute "age" of a scalar would raise an AttributeError exception.
- This approach seems to just use the necessary data for the calculation, and as we'll see it is orders of magnitude faster.

```
%timeit -n1000 titanic['is_old'] = titanic.apply(is_old_func, axis='columns')
%timeit -n1000 titanic['is_old'] = titanic.age.apply(is_old_func_series)
Out[]:
1000 loops, best of 3: 22.3 ms per loop
1000 loops, best of 3: 458 µs per loop
```

view raw

In the presence of "a lot" of rows (more than 10.000) and simple calculations for the new column, there's an alternative whose execution time is orders of magnitude lower than *Series.apply's.* This alternative is the *eval* method.

eval is extremely fast in this case, because it performs arithmetic and boolean expression evaluations "all at once" using the underlying engine **numexpr**, which is a high performance numerical evaluator for NumPy.

Its syntax is a little more friendly than that of *apply*, but for now does not support **if operations**. To do assignments to a subset of the DataFrame you could use a combination of *loc/ix* and *eval.*

```
titanic.eval('is_old = age > 60', inplace=True)
```

view raw

Reshaping

Reshaping is, broadly speaking transforming the structure of the data to make it suitable for further analysis. In the context of Pandas, we can reshape a DataFrame by using one column's values as the **index,** and another column's values as **new columns**, this is called pivoting.

Pivoting

There are two main ways to apply pivoting in Pandas, the *pivot* and *pivot_table* methods.

The *pivot* function is more restrictive than *pivot_table* since it needs the DataFrame's column set as "index" to have **unique** values only.

Our data doesn't fit the *pivot* input quite properly, which is "stacked" or "record" formatted data (as indicated in the **Pandas docs**), but for the sake of demonstrating its usage, we'll tweak our titanic DataFrame a little bit.

```python
p_titanic = titanic.drop_duplicates('age').pivot(index='age', columns='class', values='fare')
p_titanic.tail(3)
```

view raw

class	First	Second	Third
age			
71.0	34.6542	NaN	NaN
74.0	NaN	NaN	7.775
80.0	30.0000	NaN	NaN

We first removed all rows with duplicated "age" values (keeping the first appearance of each one) and pivoted that DataFrame. The **index** of that new DataFrame are the labels in the "age" column (all unique now), the **columns** of the new DataFrame are the unique **labels** of the "class" column in the original DataFrame, and the values are the corresponding "fare" values.

A better alternative for this is to use the *pivot_table* method, since it saves us from having to drop duplicates (which in this case makes no sense), and it aggregates the DataFrame values (by default performs an average).

```python
pt_titanic = titanic.pivot_table(index='age', columns='class', values='fare')
pt_titanic.tail(3)
```

view raw

class	First	Second	Third
age			
71.0	42.0792	NaN	NaN
74.0	NaN	NaN	7.775
80.0	30.0000	NaN	NaN

In the *pivot* example we have only **one row** with “age” == 71 and “class” == “First”, therefore the value (34.6542) is the “fare” value of **that row**. Whilst in the *pivot_table* example, we have **several rows** with “age” == 71 and “class” == “First”, and the value (42.0792) is the average of the “fare” values of **those rows**.

Something cool we could calculate, for example, is the median of the paid fares per embarking town, per age group. We can do this easily with *pivot_table* in one line. We need to indicate that the aggregating function to be used is *np.median.*

```python
titanic.pivot_table(index='embark_town', columns='age_group', values='fare', aggfunc=np.median
```

view raw

The inverse process of pivoting, unpivoting we might call it, is implemented in Pandas by the *pandas.melt* function.

This function takes a DataFrame and “melts it”, that is, it takes one or more columns and uses them as “indentifier variables” (keeps them the way they are). The rest of the columns are “converted” into two “unidentifier (or measured) variable” columns **“variable”** and **“value”**.

```python
pd.melt(
    p_titanic.reset_index(),
    id_vars='age',
    var_name='class_renamed',
    value_vars=['First', 'Second', 'Third'],
    value_name='fare'
).tail(3)
```

view raw

	age	class_renamed	fare
264	71.0	Third	NaN
265	74.0	Third	7.775
266	80.0	Third	NaN

In the example:

- **p_titanic.reset_index()**: We do this, because p_titanic does not have a column named "age" with the age values, but an index named "age" with the age values, and for the unpivoting process we're going to want to have the "age" column.
- **id_vars='age'**: This indicates that our "identifying variable" is going to be the column "age" (we can have more than one identifying variable/column when melting)
- **var_name:** This argument simply sets the name for the **"variable"** column. We want it to be named "class_renamed". If None, the p_titanic.reset_index().columns.name ("class" in this case) value is used, and if that's not set, the column will just be named "variable".
- **value_vars**: These are the columns that are going to be "unpivoted" to the rows axis, and are going to be values of the "variable" column. If not set, the method considers all non identifying variable columns.
- **value_name:** This simply sets the name for the **"value"** column. We want it to be named "fare". If None, the name "value" is used.

Stacking

We can also reshape a DataFrame by stacking/unstacking. This is related to *Hierarchical indexes* or MultiIndexes, which are a way to represent higher (than 1 or 2) dimensional data in Series or DataFrames.

If you recall the groupby snippet where we called *DataFrameGroupBy.size,* the result of that invocation was a Series with a MultiIndex with two levels, the first one (level 0) corresponding to the "age_group", and the second one (level 1) corresponding to the "alone" flag.

You can see how MultiIndexes work very easily, simply by removing the "values" argument in the *pivot/pivot_table* examples. This will create a DataFrame with

MultiIndexed columns with two levels; the first one being the value (“fare”, “age”, “survived”, etc.) and the second one, the “age_group”. Therefore, you can get an equivalent result with the following line:

```python
titanic.pivot_table(index='embark_town', columns='age_group', aggfunc=np.median).fare
```
view raw

But back to the point, stacking and unstacking. **Stacking** is to take a level of the columns index and transfer it to the rows index. Therefore, unstacking is the opposite; transferring a level from the rows level to the columns level.

This works even with DataFrames with single level axes. Stacking or unstacking a DataFrame of this characteristics would turn it into a MultiIndex Series!

We can take the MultiIndex Series from the groupby snippet mentioned earlier, and unstack any of its levels.

```python
groups.size().unstack()
```
view raw

alone	False	True
age_group		
child	67	2
teenager	23	21
adult	216	363
elder	4	18

By default, *stack/unstack* takes the **highest level**, in this case the level 1, but we could unstack the level 0

```python
groups.size().unstack(level=0)
```
view raw

age_group	child	teenager	adult	elder
alone				

age_group	child	teenager	adult	elder
False	67	23	216	4
True	2	21	363	18

stack works analogously.

TimeSeries

Last but not least, Pandas offers a lot of high level methods to work with TimeSeries. These methods are as powerful as they're useful, and they deserve a blog post on their own.

Nevertheless, they shouldn't pass unnoticed in this overview. TimeSeries are basically Series where the index is of a special type (DateTimeIndex), and the labels are timestamps!

With Pandas, you can resample these TimeSeries (upsample or downsample), generate Date ranges, perform non-standard increments (e.g. shift X number of business days), and much more.

We recommend you check out the **TimeSeries section** in the Pandas docs whenever you need a reference, as well as reading some of the great blog posts out there, where you'll see most of these features applied in cool and meaningful ways to real data.

Seaborn

If you've gotten this far, first of all congratulations . Secondly, don't worry, we won't be giving you an extensive description of Seaborn's features. Instead we'll just go over what it is, some of its benefits, and show you some cool plots you can make with it.

Seaborn is a Python library for making statistical visualizations. It's built to provide eye candy plots and at the same time it makes developers' life easier. We've all been in the situation where we needed to build a simple decent looking histogram and ended up making several function invocations and setting arguments without fully knowing what we're doing.

Seaborn tackles this not by reinventing the wheel but by improving it. It is built on top of Matplotlib and provides a high level API that makes "a well-defined set of hard things easy" (as stated in the **docs**), amongst other things by making that its methods work greatly by passing a minimal set of arguments.

The eye candy comes from the built in themes, the possibility to build custom attractive color palettes, and the witty way they're utilized to display statistical sugar (e.g. the kernel density estimation in a **violin plot**)

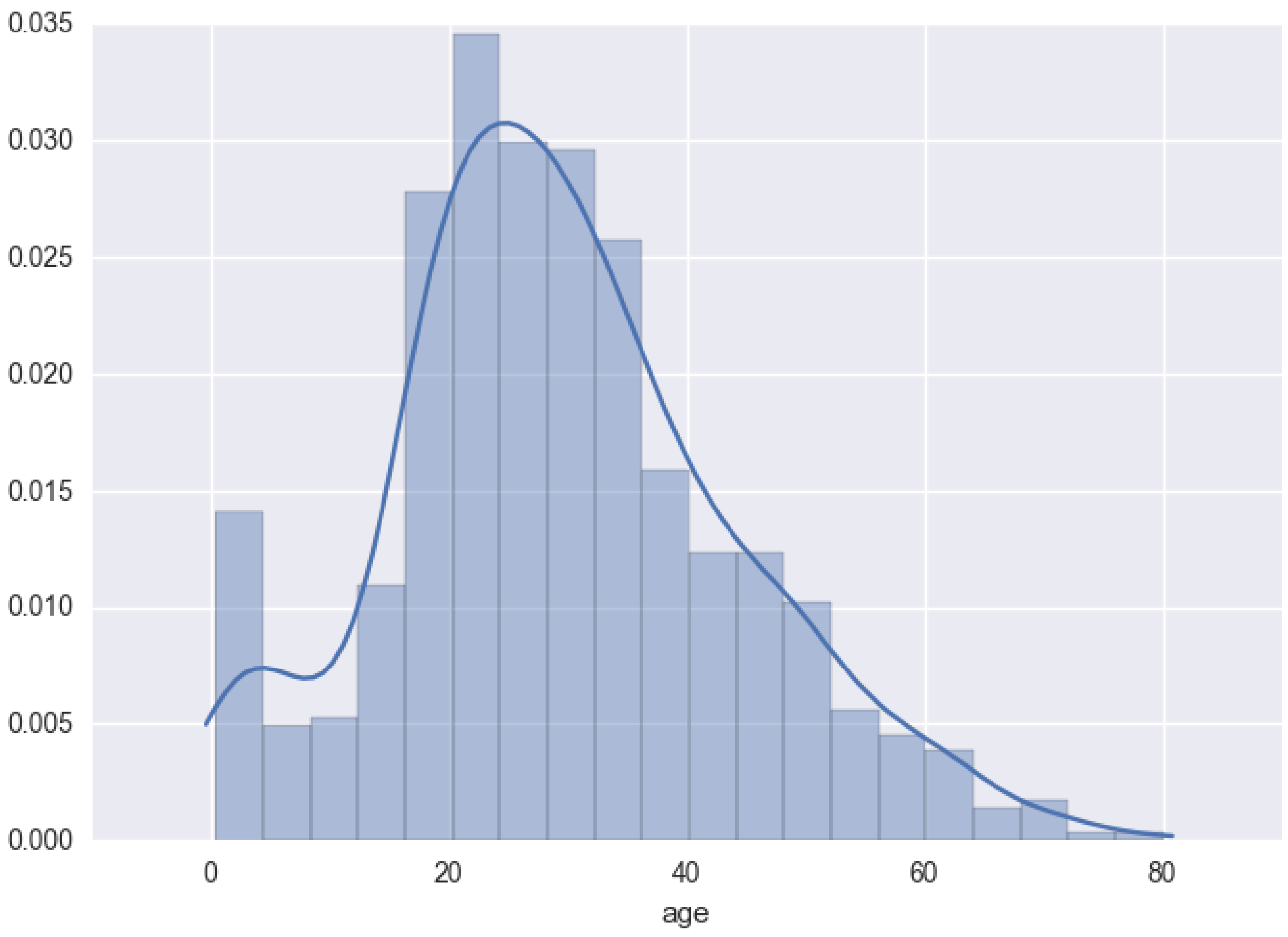
Seaborn is part of the PyData stack, and accepts Pandas' data structures as inputs in its API (thank goodness)

Let get to the plots!

- ***distplot***: The first thing you want to see when exploring your data is the distribution of your variables. For example, let's see the Titanic's passengers' ages distribution

```
sns.distplot(titanic.age.dropna())
sns.plt.show()
```

view raw



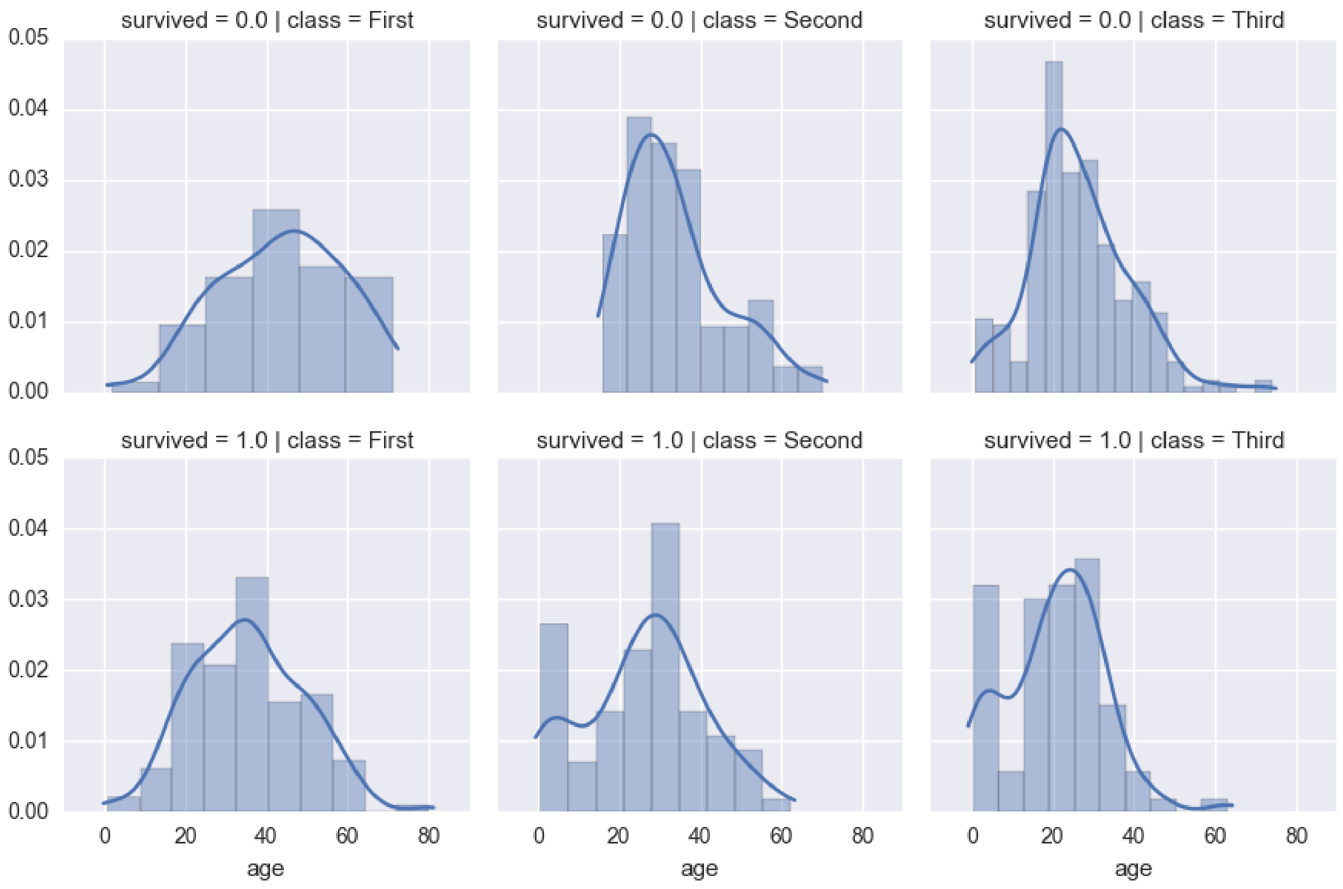
Distribution of Titanic's passengers' ages.

 - If we want to see the raw number of rows in each bin, we can pass ***kde=False*** (kernel density estimation = False)
 - We need to drop NaN values for *distplot* not to raise a ValueError exception.

- *FacetGrid:* If we wanted to break down a plot (e.g. the last one) by some categories, we needn't perform boolean queries, nor groupbys, we can use *FacetGrid.*

```python
g = sns.FacetGrid(titanic, row='survived', col='class')
g.map(sns.distplot, "age")
sns.plt.show()
```

view raw



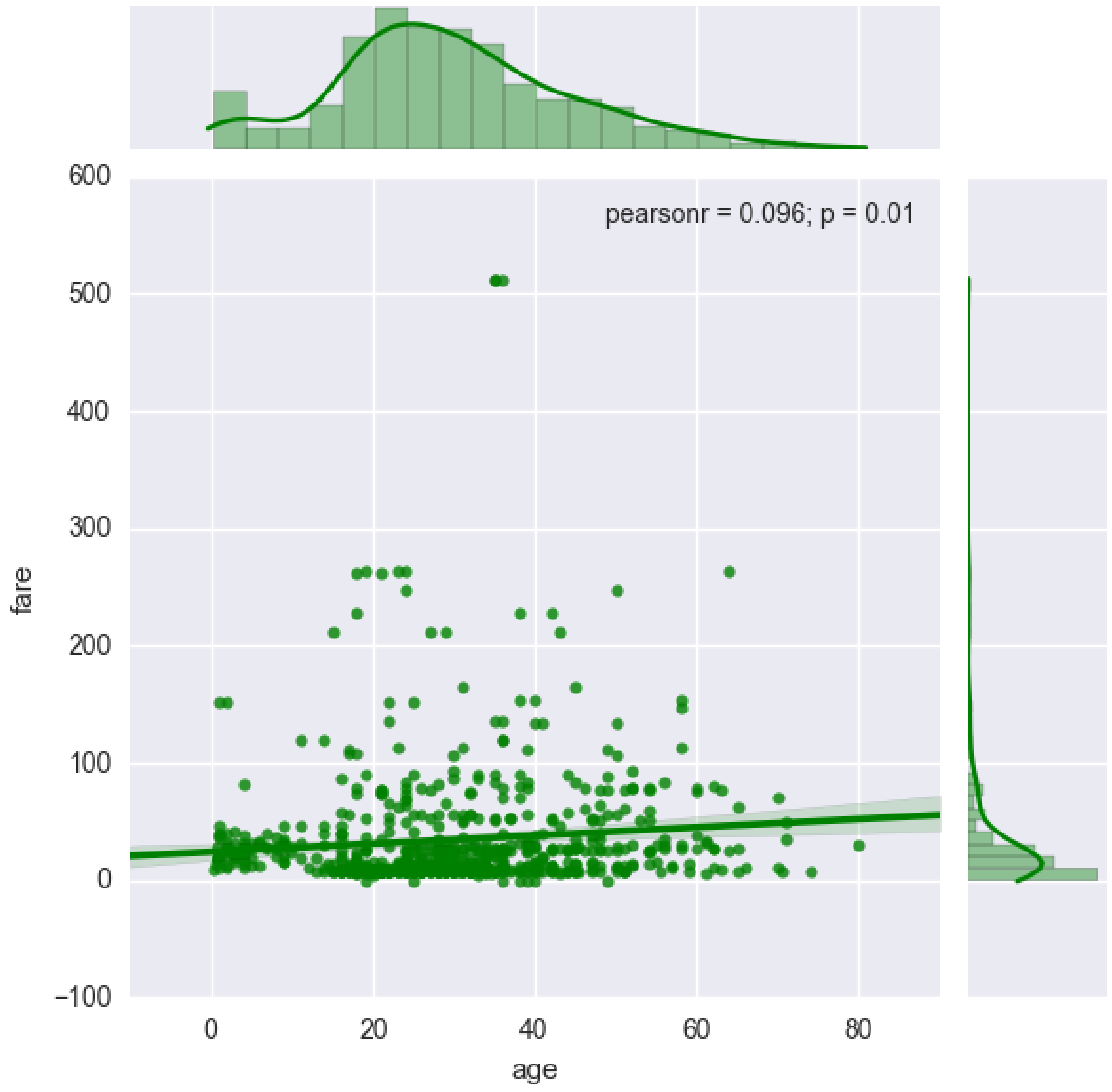
Distribution of ages in groups of passengers

- *jointplot:* This method is used to display data points according to two variables, along with both their distributions, kernel density estimators, and an optional regression that fits the data. With "reg" we indicate that we want a regression fit to the data.

- In this case, although there appears to be a small tendency upwards shown by the regression, there appears to be almost no correlation between the variables “age” and “fare”, as shown by the **Pearson** correlation coefficient.

```
sns.jointplot(data=titanic, x='age', y='fare', kind='reg', color='g')
sns.plt.show()
```

view raw



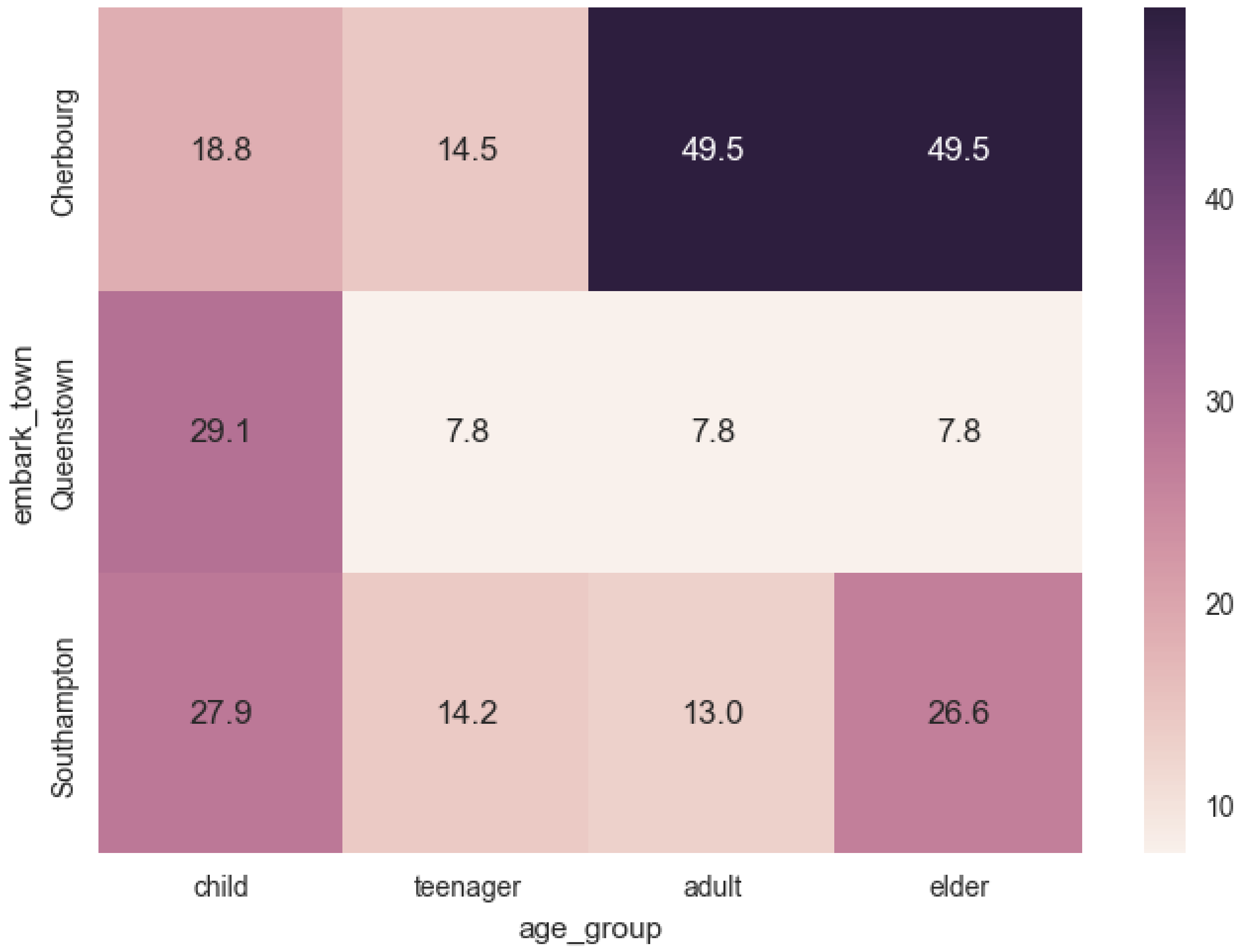
Data points by their "age" and "fare" values, along with the independent variable distributions

- ***heatmap***: Heatmaps are ideal to plot “rectangular data” such as matrixes. They’re great to visualize when some values, or calculated values, such as averages, counts, etc. are more extreme.
  - We can take the pt_titanic DataFrame from the *pivot_table,* which held data of the median fares paid by passengers per embark_town per age_group,

and build a heatmap very easily. Most times, we like out heatmaps annotated to catch some subtelties that may pass by me with the colors. The “fmt” value is pretty straightforward.

```
df = titanic.pivot_table(index='embark_town', columns='age_group', values='fare', aggfunc
sns.heatmap(df, annot=True, fmt=".1f")
```

view raw

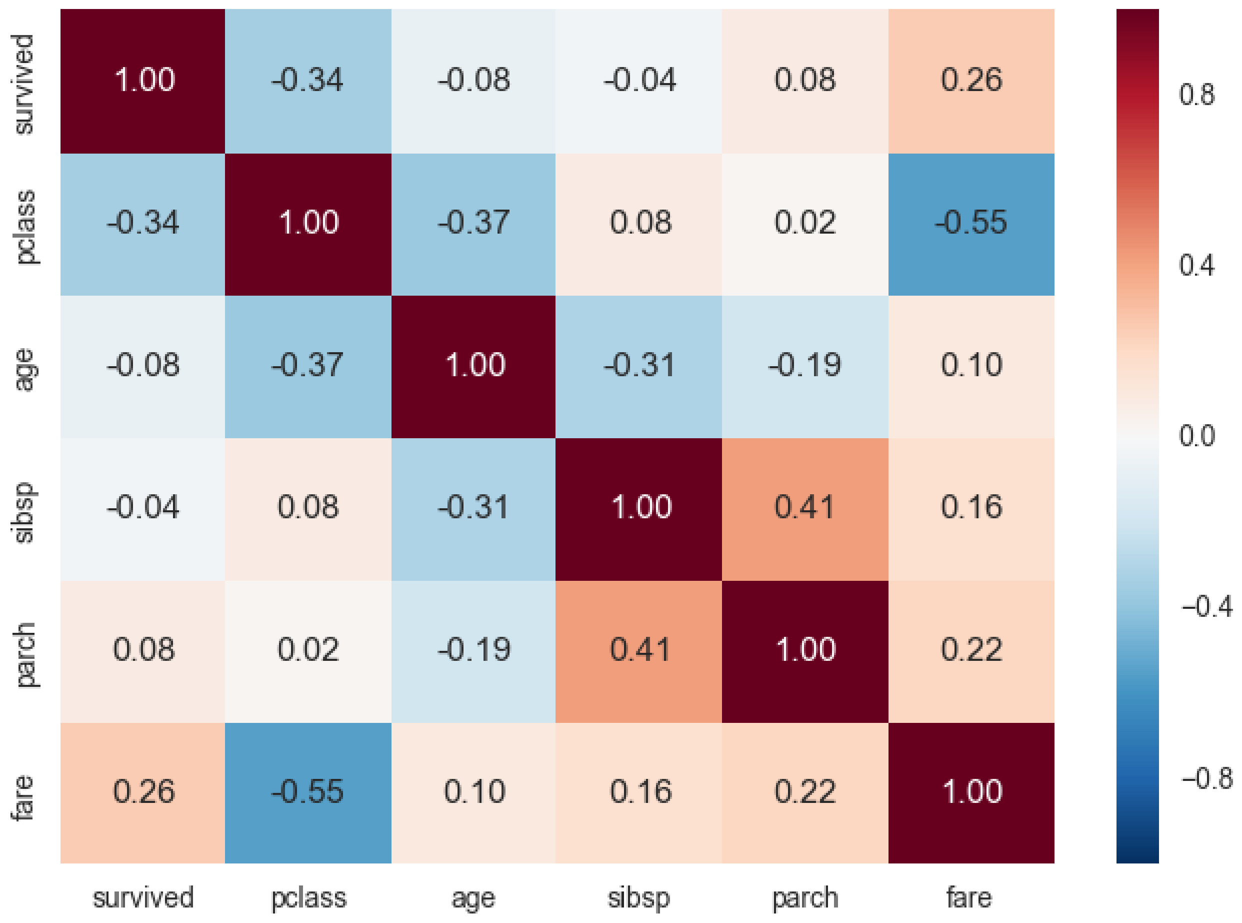


Annotated heatmap of the paid fares' median by groups of passengers

- Finally, something really cool that you can put into a heatmap is a correlation matrix. Pandas DataFrame has a *corr* method that calculates Pearson’s (can be another) correlation coefficient between all couples of numeric columns of the DataFrame.

```
sns.heatmap(titanic.corr(), annot=True, fmt=".2f")
```

view raw



Annotated heatmap of Pearson correlation coefficients between variables

This's it! Hopefully this post was useful for you to get to know Pandas in a gradual and ordered way, and motivates you to dig deeper into the features without fear. With much less detail, the post also attempted to give a glance at Seaborn's beautiful graphs; we highly encourage you to dive into the **Seaborn API** and the tutorials so as to really get acquainted with it.

Have fun using this tools to better analyze and explore data!

Update (2017-03-21): *In a previous version of this post, the "is_old" assignments were done using dot notation. As indicated in the comments, using dot notation carelessly for this can cause an undesired overwrite of DataFrame attributes for that object. But even more so, the assignment of a* ***new column*** *this way* ***does not work****. What works is accessing/updating an existing column, which is why we probably missed this in the first place.*

At ***Tryolabs*** *we are specialists in building Python Backends with Machine Learning components which leverage large amounts of data. If you need some help in these kind of* *projects, drop us a line to **hello@tryolabs.com** (**or fill out this form**) and we’ll happily connect.*
