Pivot Tables

We have seen how the GroupBy abstraction can be used to delve deeper into a dataset and discover hidden patterns. A pivot table performs a similar function and is widely used in spreadsheets and other tabular data processing software. Data organized in columns is taken as input by the pivot table, which then groups the entries into a two-dimensional table that summarizes the data in multiple dimensions. Some people get confused between pivot tables and GroupBy, but I find it useful to think of pivot tables as essentially a multidimensional version of GroupBy aggregation. That is, the steps involve a split, an application, and a combine, but instead of working across a single dimension, the data is laid out across a pair of them.

Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the *Titanic*, available through the Seaborn library.

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

This contains a wealth of information on each passenger of that ill-fated voyage, including gender, age, class, fare paid, and much more.

Pivot Tables by Hand

Grouping by gender, survival status, or both could be a good starting point for gaining insight into this data. After reading the preceding section, you might be tempted to use a GroupBy operation; for instance, let's analyze the survival rate in terms of male and female patients.

```
titanic.groupby('sex')[['survived']].mean()
```

	survived
sex	
female	0.742038
male	0.188908

Immediate insight is provided by the fact that three out of every four ladies on board made it, whereas only one out of every five men did.

Very helpful, however it would also be interesting to compare survival rates based on other factors, such as socioeconomic status. GroupBy terminology suggests the following steps: group by class and gender, choose survivors, apply a mean aggregate, combine the resultant groups, and then unstack the hierarchical index to uncover the latent multidimensionality. secret code

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

class	**First**	**Second**	**Third**
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularly easy to read or use. This two-dimensional `GroupBy` is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multi-dimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the preceding operation using the `pivot_table` method of `DataFrame`s:

```
titanic.pivot_table('survived', index='sex', columns='class')
```

class	**First**	**Second**	**Third**
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

This is eminently more readable than the `groupby` approach, and produces the same result. As you might expect of an early 20th-century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women survived with near certainty (hi, Rose!), while only one in ten third-class men survived (sorry, Jack!).

Multi-level pivot tables

Just as in the `GroupBy`, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the `pd.cut` function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

	class	**First**	**Second**	**Third**
sex	**age**			
female	**(0, 18]**	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	**(0, 18]**	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

We can apply the same strategy when working with the columns as well; let's add info on the fare paid using `pd.qcut` to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

	fare	**[0, 14.454]**			**(14.454, 512.329]**		
	class	**First**	**Second**	**Third**	**First**	**Second**	**Third**
sex	**age**						
female	**(0, 18]**	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.391304
male	**(0, 18]**	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 80]	0.0	0.098039	0.125000	0.391304	0.030303	0.192308

Additional pivot table options

The full call signature of the `pivot_table` method of `DataFrame`s is as follows:

```
# call signature as of Pandas 0.18
DataFrame.pivot_table(data, values=None, index=None, columns=None,
                      aggfunc='mean', fill_value=None, margins=False,
                      dropna=True, margins_name='All')
```

We've already seen examples of the first three arguments; here we'll take a quick look at the remaining ones. Two of the options, `fill_value` and `dropna`, have to do with missing data and are fairly straightforward; we will not show examples of them here.

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As in the GroupBy, the aggregation specification can be a string representing one of several common choices (e.g., `'sum'`, `'mean'`, `'count'`, `'min'`, `'max'`, etc.) or a function that implements an aggregation (e.g., `np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as a dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='sex', columns='class',
                    aggfunc={'survived':sum, 'fare':'mean'})
```

	fare			**survived**		
class	**First**	**Second**	**Third**	**First**	**Second**	**Third**
sex						
female	106.125798	21.970121	16.118810	91.0	70.0	72.0
male	67.226127	19.741782	12.661633	45.0	17.0	47.0

Notice also here that we've omitted the `values` keyword; when specifying a mapping for `aggfunc`, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

class	**First**	**Second**	**Third**	**All**
sex				
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%. The margin label can be specified with the `margins_name` keyword, which defaults to `"All"`.

Vectorized String Operations

One of Python's strengths is how simple it is to work with strings. To this end, Pandas expands upon this foundation by providing a rich toolkit of vectorized string operations that are essential for the kind of munging needed when dealing with (read: cleaning up) real-world data. We'll go through some of the string operations available in Pandas and then use them to clean up a somewhat unorganized collection of recipes we gathered from the web.

Introducing Pandas String Operations

Tools like NumPy and Pandas expand arithmetic operations, as we saw in earlier parts, allowing us to rapidly and simply apply the same action to a large number of array items. To provide only one illustration:

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
array([ 4,  6, 10, 14, 22, 26])
```

By eliminating the need to specify the size and shape of an array before performing an operation on it, the syntax for working with arrays of data has been greatly simplified thanks to vectorization of operations. Unfortunately, NumPy does not give this level of convenience for arrays of strings, forcing you to resort to more verbose loop syntax:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-fc1d891ab539> in <module>()
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

<ipython-input-3-fc1d891ab539> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

```
import pandas as pd
names = pd.Series(data)
names
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

Using tab completion on this `str` attribute will list all the vectorized string methods available to Pandas.

Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Notice that these have various return values. Some, like `lower()`, return a series of strings:

```
monte.str.lower()
0    graham chapman
1       john cleese
2     terry gilliam
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

But some others return numbers:

```
monte.str.len()
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

Or Boolean values:

```
monte.str.startswith('T')
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

Still others return lists or other compound values for each element:

```
monte.str.split()
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in `re` module:

Method	Description
`match()`	Call `re.match()` on each element, returning a boolean.
`extract()`	Call `re.match()` on each element, returning matched groups as strings.
`findall()`	Call `re.findall()` on each element
`replace()`	Replace occurrences of pattern with some other string
`contains()`	Call `re.search()` on each element, returning a boolean
`count()`	Count occurrences of pattern
`split()`	Equivalent to `str.split()`, but accepts regexps
`rsplit()`	Equivalent to `str.rsplit()`, but accepts regexps

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
monte.str.extract('([A-Za-z]+)', expand=False)
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
```

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (`^`) and end-of-string (`$`) regular expression characters:

```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
0    [Graham Chapman]
1                  []
2     [Terry Gilliam]
3                  []
4       [Terry Jones]
5     [Michael Palin]
dtype: object
```

The ability to concisely apply regular expressions across `Series` or `Dataframe` entries opens up many possibilities for analysis and cleaning of data.

Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description
`get()`	Index each element
`slice()`	Slice each element
`slice_replace()`	Replace slice in each element with passed value
`cat()`	Concatenate strings
`repeat()`	Repeat values

Method	Description
`normalize()`	Return Unicode form of string
`pad()`	Add whitespace to left, right, or both sides of strings
`wrap()`	Split long strings into lines with length less than a given width
`join()`	Join strings in each element of the Series with passed separator
`get_dummies()`	extract dummy variables as a dataframe

Vectorized item access and slicing

The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python's normal indexing syntax–for example, `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar.

These `get()` and `slice()` methods also let you access elements of arrays returned by `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

```
monte.str.split().str.get(-1)
0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

Indicator variables

Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing some sort of coded indicator. For example, we might have a dataset that contains information in the form of codes, such as A="born in America," B="born in the United Kingdom," C="likes cheese," D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
full_monte
```

	info	**name**
0	B|C|D	Graham Chapman
1	B|D	John Cleese
2	A|C	Terry Gilliam
3	B|D	Eric Idle
4	B|C	Terry Jones
5	B|C|D	Michael Palin

The `get_dummies()` routine lets you quickly split-out these indicator variables into a `DataFrame`:

```
full_monte['info'].str.get_dummies('|')
```

	A	**B**	**C**	**D**
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

With these operations as building blocks, you can construct an endless range of string processing procedures when cleaning your data.