Merging Dataframes

Last week, we were introduced to the pandas data manipulation and analysis library.
We saw that there are really two-core data structures which are very similar, the one-dimensional series object and the two-dimensional DataFrame object.
Querying these two data structures is done in a few different ways, such as using the *iloc* or *loc* attributes for row-based querying, or using the square brackets on the object itself for column-based querying.
Most importantly, we saw that one can query the DataFrame and Series objects through Boolean masking.
And Boolean masking is a powerful filtering method which allows us to use broadcasting to determine what data should be kept in our analysis.

In this week's lecture, we're going to go into more detail on how to manipulate DataFrame, in particular.
We're going to explore how to reduce and process data using GroupBy and Apply, and how to join data sets from multiple files together into one.
We'll also talk about some features that pandas has which are useful for both traditional statistical analyses and machine learning.
Let's start with how to merge data sets.
We've already seen how we add new data to an existing DataFrame.
We simply use the square bracket operator with the new column name, and the data is added as long as an index is shared.
If there is no shared index and a scalar value is passed in, which remember a scalar value is just a single value like an integer or a string.
The new value column is added with the scalar value as the default value.
What if we wanted to assign a different value for every row?
Well, it gets trickier.
If we could hardcode the values into a list, then pandas will unpack them and assign them to the rows.
But if the list we have isn't long enough, then we can't do this, since Pandas doesn't know where the missing values should go.

Here's an example.
We used the DataFrame of store purchases from one of our previous lectures, where the index is a list of stores and the columns store purchase data.

```
import pandas as pd

df = pd.DataFrame([{'Name': 'Chris', 'Item Purchased': 'Sponge', 'Cost': 22.50},
                   {'Name': 'Kevyn', 'Item Purchased': 'Kitty Litter', 'Cost': 2.50},
                   {'Name': 'Filip', 'Item Purchased': 'Spoon', 'Cost': 5.00}],
                  index=['Store 1', 'Store 1', 'Store 2'])
df
```

	Name	Item Purchased	Cost
Store 1	Chris	Sponge	22.5
Store 1	Kevyn	Kitty Litter	2.5
Store 2	Filip	Spoon	5.0

If we want to add some new column called Date to the DataFrame, that's fine. We just use the square bracket operator directly on the DataFrame, as long as the column is as long as the rest of the records.
If we want to add some new field, may be a delivery flag, that's easy too since it's a scalar value.

```
df['Date'] = ['December 1', 'January 1', 'mid-May']
df
```

	Name	Item Purchased	Cost	Date
Store 1	Chris	Sponge	22.5	December 1
Store 1	Kevyn	Kitty Litter	2.5	January 1
Store 2	Filip	Spoon	5.0	mid-May

```
df['Delivered'] = True
df
```

	Name	Item Purchased	Cost	Date	Delivered
Store 1	Chris	Sponge	22.5	December 1	True
Store 1	Kevyn	Kitty Litter	2.5	January 1	True
Store 2	Filip	Spoon	5.0	mid-May	True

The problem comes in when we have only a few items to add.
In order for this to work, we have to supply pandas the list which is long enough for the DataFrame, so that each row could be populated.
This means that we have to input none values ourselves.

```
df['Feedback'] = ['Positive', None, 'Negative']
df
```

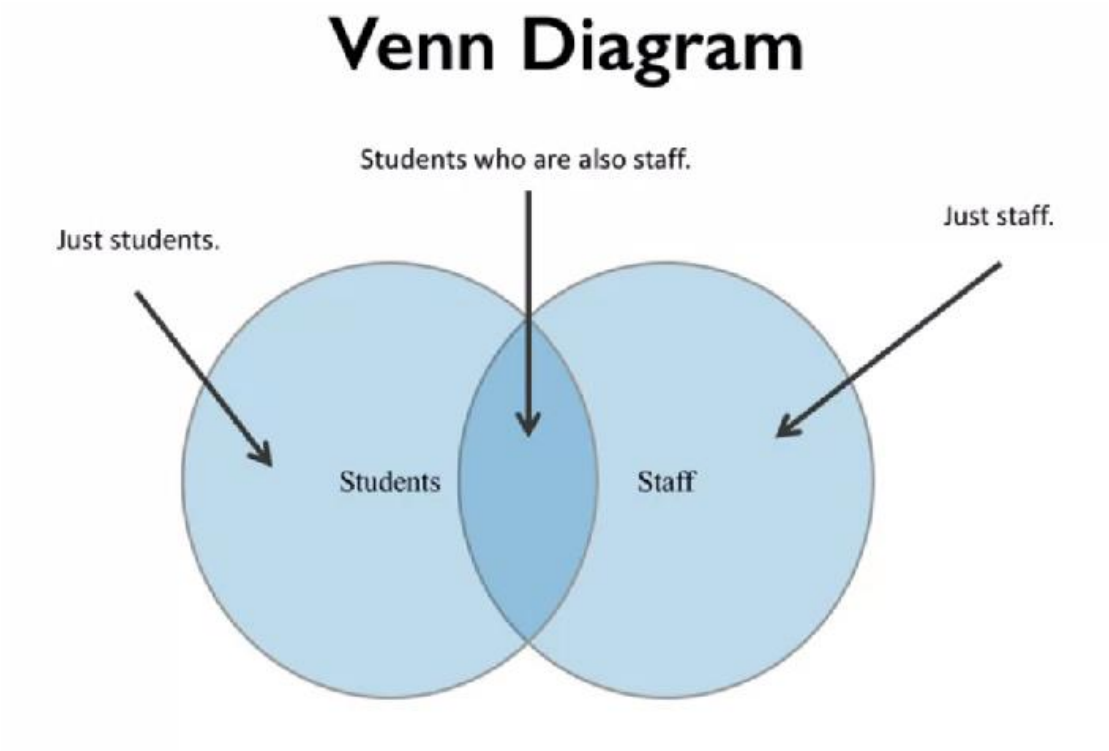
	Name	Item Purchased	Cost	Date	Delivered	Feedback
Store 1	Chris	Sponge	22.5	December 1	True	Positive
Store 1	Kevyn	Kitty Litter	2.5	January 1	True	None
Store 2	Filip	Spoon	5.0	mid-May	True	Negative

If each of our rows has a unique index, then we could just assign the new column identifier to the series.
For instance, if we reset the index in this example so the DataFrame is labeled from 1 through 2, then we create a new series with these labels, we can apply it.
And the results we get are as we expected.
The nice aspect of this approach is that we could just ignore the items that we don't know about, and pandas will put missing values in for us. So this is a really nice way to do it.

```
adf = df.reset_index()
adf['Date'] = pd.Series({0: 'December 1', 2: 'mid-May'})
adf
```

	index	Name	Item Purchased	Cost	Date	Delivered	Feedback
0	Store 1	Chris	Sponge	22.5	December 1	True	Positive
1	Store 1	Kevyn	Kitty Litter	2.5	NaN	True	None
2	Store 2	Filip	Spoon	5.0	mid-May	True	Negative

More commonly, we want to join two larger DataFrames together, and this is a bit more complex.
Before we jump into the code, we need to address a little relational theory, and to get some language conventions down.
This is a Venn diagram.



A Venn diagram is traditionally used to show set membership.
For example, the circle on the left is the population of students at a university.
The circle on the right is the population of staff at a university.
And the overlapping region in the middle are all of those students who are also staff.

Maybe these students run tutorials for a course, or grade assignments, or engage in running research experiments.
We could think of these two populations as indices in separate DataFrames, maybe with the label of Person Name.

When we want to join the DataFrames together, we have some choices to make.
First what if we want a list of all the people regardless of whether they're staff or student, and all of the information we can get on them?
In database terminology, this is called a full outer join. And in set theory, it's called a union. In the Venn diagram, it represents everyone in any circle.

It's quite possible though that we only want those people who we have maximum information for, those people who are both staff and students.
In database terminology, this is called an inner join. Or in set theory, the intersection.
And this is represented in the Venn diagram as the overlapping parts of each circle.

Okay, so let's see an example of how we would do this in pandas, where we would use the merge function.
First we create two DataFrames, staff and students.

There's some overlap in these DataFrames, in that James and Sally are both students and staff, but Mike and Kelly are not.
Importantly, both DataFrames are indexed along the value we want to merge them on, which is called Name.

```
staff_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR'},
                         {'Name': 'Sally', 'Role': 'Course liasion'},
                         {'Name': 'James', 'Role': 'Grader'}])
staff_df = staff_df.set_index('Name')
student_df = pd.DataFrame([{'Name': 'James', 'School': 'Business'},
                           {'Name': 'Mike', 'School': 'Law'},
                           {'Name': 'Sally', 'School': 'Engineering'}])
student_df = student_df.set_index('Name')
print(staff_df.head())
print()
print(student_df.head())
```

```
                Role
Name
Kelly  Director of HR
Sally  Course liasion
James          Grader

            School
Name
James     Business
Mike           Law
Sally  Engineering
```

If we want the union of these, we would call merge passing in the DataFrame on the left and the DataFrame on the right, and telling merge that we want it to use an outer join.
We tell merge that we want to use the left and right indices as the joining columns.

```
pd.merge(staff_df, student_df, how='outer', left_index=True, right_index=True)
```

Name	Role	School
James	Grader	Business
Kelly	Director of HR	NaN
Mike	NaN	Law
Sally	Course liasion	Engineering

We see in the resulting DataFrame that everyone is listed.
And since Mike does not have a role, and John does not have a school, those cells are listed as missing values.
If we wanted to get the intersection, that is, just those students who are also staff, we could set the how attribute to inner.
And we set the resulting DataFrame has only James and Sally in it.

```
pd.merge(staff_df, student_df, how='inner', left_index=True, right_index=True)
```

Name	Role	School
Sally	Course liasion	Engineering
James	Grader	Business

Now there are two other common use cases when merging DataFrames.
Both are examples of what we would call set addition.
The first is when we would want to get a list of all staff regardless of whether they were students or not.
But if they were students, we would want to get their student details as well.
To do this we would use a left join.

```
pd.merge(staff_df, student_df, how='left', left_index=True, right_index=True)
```

Name	Role	School
Kelly	Director of HR	NaN
Sally	Course liasion	Engineering
James	Grader	Business

You could probably guess what comes next.
We want a list of all of the students and their roles if they were also staff.
To do this we would do a right join.

```
pd.merge(staff_df, student_df, how='right', left_index=True, right_index=True)
```

Name	Role	School
James	Grader	Business
Mike	NaN	Law
Sally	Course liasion	Engineering

The merge method has a couple of other interesting parameters.
First, you don't need to use indices to join on; you can use columns as well.
Here's an example.

```
staff_df = staff_df.reset_index()
student_df = student_df.reset_index()
pd.merge(staff_df, student_df, how='left', left_on='Name', right_on='Name')
```

	Name	Role	School
0	Kelly	Director of HR	NaN
1	Sally	Course liasion	Engineering
2	James	Grader	Business

So what happens when we have conflicts between the DataFrames?
Let's take a look by creating new staff and student DataFrames that have a location information added to them.
In the staff DataFrame, this is an office location where we can find the staff person.
And we can see the Director of HR is on State Street, while the two students are on Washington Avenue.
But for the student DataFrame, the location information is actually their home address.

```
staff_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR', 'Location': 'State Street'},
                         {'Name': 'Sally', 'Role': 'Course liasion', 'Location': 'Washington Avenue'},
                         {'Name': 'James', 'Role': 'Grader', 'Location': 'Washington Avenue'}])
student_df = pd.DataFrame([{'Name': 'James', 'School': 'Business', 'Location': '1024 Billiard Avenue'},
                           {'Name': 'Mike', 'School': 'Law', 'Location': 'Fraternity House #22'},
                           {'Name': 'Sally', 'School': 'Engineering', 'Location': '512 Wilson Crescent'}])
pd.merge(staff_df, student_df, how='left', left_on='Name', right_on='Name')
```

	Name	Role	Location_x	School	Location_y
0	Kelly	Director of HR	State Street	NaN	NaN
1	Sally	Course liasion	Washington Avenue	Engineering	512 Wilson Crescent
2	James	Grader	Washington Avenue	Business	1024 Billiard Avenue

The merge function preserves this information, but appends an _x or _y to help differentiate between which index went with which column of data.
The _x is always the left DataFrame information, and
the _ y is always the right DataFrame information.
And you could control the names of _x and _y with additional parameters if you want to.
Now you try it.

Before we leave merging of DataFrames, let's talk about multi-indexing and multiple columns.
It's quite possible that the first name for students and staff might overlap, but the last name might not.
In this case, we use a list of the multiple columns that should be used to join keys on the left_on and right_on parameters.
As you see here, James Wilde and James Hammond don't match on both keys.

So the inner join doesn't include these individuals in the output, and only Sally Brooks is retained.

```
staff_df = pd.DataFrame([{'First Name': 'Kelly', 'Last Name': 'Desjardins', 'Role': 'Director of HR'},
                         {'First Name': 'Sally', 'Last Name': 'Brooks', 'Role': 'Course liasion'},
                         {'First Name': 'James', 'Last Name': 'Wilde', 'Role': 'Grader'}])
student_df = pd.DataFrame([{'First Name': 'James', 'Last Name': 'Hammond', 'School': 'Business'},
                           {'First Name': 'Mike', 'Last Name': 'Smith', 'School': 'Law'},
                           {'First Name': 'Sally', 'Last Name': 'Brooks', 'School': 'Engineering'}])
staff_df
student_df
pd.merge(staff_df, student_df, how='inner', left_on=['First Name','Last Name'], right_on=['First Name','Last Name'])
```

	First Name	Last Name	Role	School
0	Sally	Brooks	Course liasion	Engineering

That's it for merging of DataFrames.
pandas has a lot more options in the area, but I think the merge function is perhaps the most easiest to understand and the most flexible.
In the next section, we're going to talk about how to write idiomatic pandas and delve a bit deeper into more advanced functions for manipulating DataFrames.