Creating columns

`.withColumn()` method, which takes two arguments. First, a string with the name of your new column, and second the new column itself.

Spark DataFrame is *immutable*. This means that it can't be changed, and so columns can't be updated in place.

Thus, all these methods return a new DataFrame. To overwrite the original DataFrame you must reassign the returned DataFrame using the method like so:

```
df = df.withColumn("newCol", df.oldCol's operation)
```

```
# Create the DataFrame flights
flights = spark.table('flights')

# Show the head
print(flights.show())

# Add duration_hrs
flights =
flights.withColumn('duration_hrs',flights.air_ti
me/60 )

SELECT tain, des FROM flights
WHERE air_time/60 > 10;

# Filter flights with a SQL string
long_flights1 = flights.filter("distance > 1000")

# Filter flights with a boolean column
long_flights2 = flights.filter(flights.distance >
1000)

# Examine the data to check they're equal
print(long_flights1.show())
```

```
print(long_flights2.show())
```

The difference between `.select()` and `.withColumn()` methods is that `.select()` returns only the columns you specify, while `.withColumn()` returns all the columns of the DataFrame in addition to the one you defined.

# Select the first set of columns
selected1 = flights.select('tailnum','origin','dest')

# Select the second set of columns
temp = flights.select(flights.origin, flights.dest, flights.carrier)

# Define first filter
filterA = flights.origin == "SEA"

# Define second filter
filterB = flights.dest == "PDX"

# Filter the data, first by filterA then by filterB
selected2 = temp.filter(filterA).filter(filterB)

`.select()` method to perform column-wise operations. When you're selecting a column using the `df.colName` notation, you can perform any column operation and the `.select()` method will return the transformed column. For example,

```
flights.select(flights.air_time/60)
```

returns a column of flight durations in hours instead of minutes. You can also use the `.alias()` method to rename a column you're selecting. So if you wanted to `.select()` the column `duration_hrs` (which isn't in your DataFrame) you could do

```
flights.select((flights.air_time/60).alias("duration_hrs"))
```

The equivalent Spark DataFrame method `.selectExpr()` takes SQL expressions as a string:

```
flights.selectExpr("air_time/60 as duration_hrs")
```

with the SQL `as` keyword being equivalent to the `.alias()` method. To select multiple columns, you can pass multiple strings.

Define avg_speed
avg_speed =
(flights.distance/(flights.air_time/60)).alias("avg_speed")

# Select the correct columns
speed1 = flights.select("origin", "dest", "tailnum", avg_speed)

# Create the same table using a SQL expression
speed2 = flights.selectExpr("origin", "dest", "tailnum",
"distance/(air_time/60) as avg_speed")

Aggregating

All of the common aggregation methods, like `.min()`, `.max()`, and `.count()` are `GroupedData` methods. These are created by calling the `.groupBy()` DataFrame method. You'll learn exactly what that means in a few exercises. For now, all you have to do to use these functions is call that method on your DataFrame. For example, to find the minimum value of a column, `col`, in a DataFrame, `df`, you could do

```
df.groupBy().min("col").show()
```

This creates a `GroupedData` object (so you can use the `.min()`method), then finds the minimum value in `col`, and returns it as a DataFrame.

# Find the shortest flight from PDX in terms of distance
flights.filter(flights.origin == 'PDX').groupBy().min('distance').show()

Find the longest flight from SEA in terms of duration

flights.filter(flights.origin == 'SEA').groupBy().max('air_time').show()

# Average duration of Delta flights
flights.filter(flights.carrier=='DL').filter(flights.origin=='SEA').groupBy().avg('air_time').show()

# Total hours in the air
flights.withColumn("duration_hrs", flights.air_time/60).groupBy().sum('duration_hrs').show()

Grouping and Aggregating I

PySpark has a whole class devoted to grouped data frames: `pyspark.sql.GroupedData`, which you saw in the last two exercises.

You've learned how to create a grouped DataFrame by calling the `.groupBy()` method on a DataFrame with no arguments.

Now you'll see that when you pass the name of one or more columns in your DataFrame to the `.groupBy()` method, the aggregation methods behave like when you use a `GROUP BY` statement in a SQL query!

Group by tailnum

by_plane = flights.groupBy("tailnum")

Number of flights each plane made

by_plane.count().show()

```
# Group by origin

by_origin = flights.groupBy("origin")


# Average duration of flights from PDX and SEA

by_origin.avg("air_time").show()
```

Grouping and Aggregating II

```
# Import pyspark.sql.functions as F

import pyspark.sql.functions as F


# Group by month and dest

by_month_dest = flights.groupBy('month','dest')


# Average departure delay by month and destination

by_month_dest.avg('dep_delay').show()


# Standard deviation

by_month_dest.agg(F.stddev('dep_delay')).show()
```

Joining

A join will combine two different tables along a column that they share. This column is called the *key*.

Joining II

In PySpark, joins are performed using the DataFrame method `.join()`. This method takes three arguments. The first is the second DataFrame that you want to join with the first one. The second argument, `on`, is the name of the key column(s) as a string. The names of the key column(s) must be the same in each table. The third argument, `how`, specifies the kind of join to perform. In this course we'll always use the value `how="leftouter"`

```
# Examine the data
print(airports.show())

# Rename the faa column
airports = airports.withColumnRenamed('faa','dest')

# Join the DataFrames
flights_with_airports = flights.join(airports,on='dest',how='leftouter')

# Examine the data again
print(flights_with_airports.show())
```