Data Science Bootcamp at Spark Summit Analyze Data and Build a Dashboard with PixieDust

San Francisco, June 2017

David Taieb

Distinguished Engineer

Developer Advocacy

IBM Watson Data Platform

@DTAIEB55

What you will learn in this Workshop

- How to use Jupyter Notebooks to load, visualize and analyze data
- IBM Data Science Experience (DSX)
- PixieDust open source Python Library
- How to build a dashboard using PixieApps
- San Francisco Open Data
- Mapbox GL

Info before we start

- The tutorial can be followed from a local Jupyter Notebook environment. However, the instructions and screenshots here walk through the notebook in the DSX environment.
- A corresponding notebook is available here: http://ibm.biz/PixieDustBootcampSFNotebook

For best results, use the latest version of either Mozilla Firefox or Google Chrome.

DSX

DSX is an interactive, collaborative, cloud-based environment where data scientists, developers, and others interested in data science can use tools (e.g., RStudio, Jupyter Notebooks, Spark, etc.) to collaborate, share, and gather insight from their data

Sign Up

- DSX is powered by IBM Bluemix, therefore your DSX login is same as your IBM Bluemix login. If you already have a Bluemix account or previously accessed DSX you may proceed to the Sign In section. Otherwise, you first need to sign up for an account.
- From your browser:
    1. Go to the DSX site: http://datascience.ibm.com
    2. Click on Sign Up
    3. Enter your Email
    4. Click Continue
    5. Fill out the form to register for IBM Bluemix

Sign In

From your browser:

1. Go to the DSX site: http://datascience.ibm.com
2. Click on Sign In
3. Enter your IBMid or email
4. Click Continue
5. Enter your Password
6. Click Sign In

Create a blank new notebook

You will need to create a noteboook to experiment with the data and a project to house your notebook.
After signing into DSX:

1. On the upper right of the DSX site, click the + and choose Create project.
2. Enter a Name for your project
3. Select a Spark Service
4. Click Create

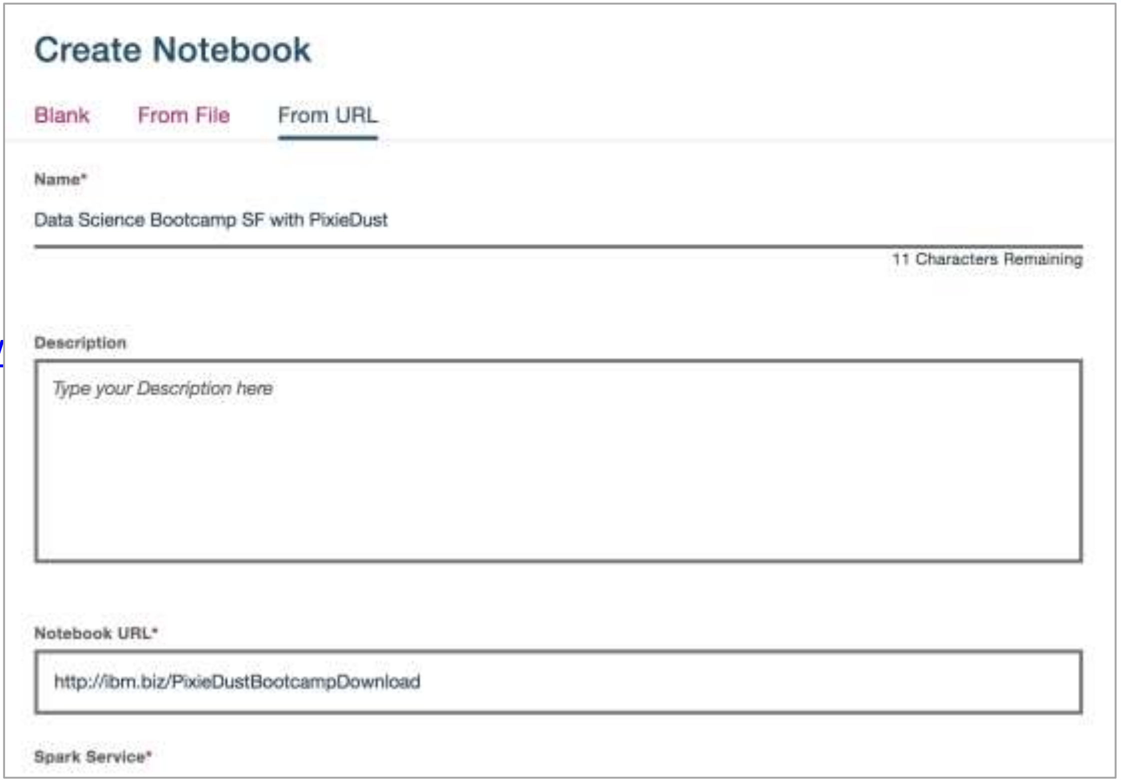
From within the new project, you will create your notebook:

1. Click add notebooks
2. Click the Blank tab in the Create Notebook form
3. Enter a Name for the notebook
4. Select Python 2 for the Language
5. Select 1.6 for the Spark version
6. Select the Spark Service
7. Click Create Notebook

You can also import the complete notebook

Skip this page if you prefer to start from a blank notebook.

1. Click add notebooks
2. Click the From URL tab in the Create Notebook form
3. Enter a Name for the notebook
4. In the Notebook URL enter: http://ibm.biz/PixieDustBootcampDownload
5. Click Create Notebook



Create Notebook

Name*

Data Science Bootcamp SF with PixieDust

11 Characters Remaining

Description

Type your Description here

Notebook URL*

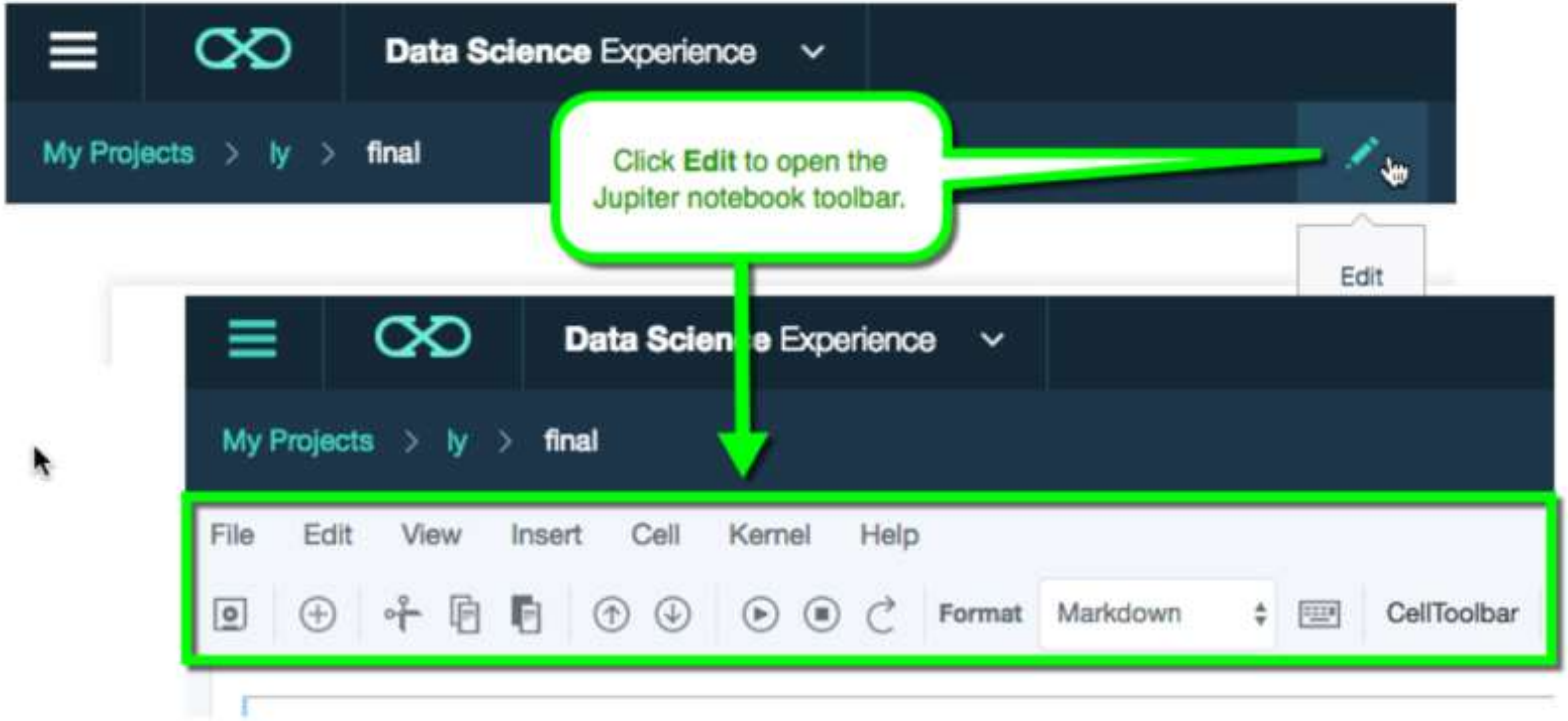
http://ibm.biz/PixieDustBootcampDownload

Spark Service*

Make sure your Notebook is in Edit Mode

- When you use a notebook in DSX, you can run a cell only by selecting it, then going to the toolbar and clicking on the Run Cell (▶) button. When a cell is running, an [*] is shown beside the cell. Once the cell has finished the asterisks is replaced by a number.
- If you don’t see the Jupyter toolbar showing the Run Cell (▶) button and other notebook controls, you are not in edit mode. Go to the dark blue toolbar above the notebook and click the edit (pencil) icon.

Edit Mode


Data Science Experience
My Projects > ly > final
Edit
File Edit View Insert Cell Kernel Help
Format Markdown CellToolbar

Click **Edit** to open the Jupiter notebook toolbar.

PixieDust

You can find detailed info on PixieDust here: http://ibm.biz/wdp-pixiedust

In this section, we'll show how to use the PixieDust Open Source Python library within a Notebook to:

Import data	Visualize your data	Explore your data	Build dashboards
Import your own data set or choose from multiple sample data sets	Visualize your data in tables, charts, maps and more	Explore and analyze your data in an interactive interface	Build dashboards and management tools with PixieApps

@DTAIEB55

Make sure PixieDust is at the latest version

- DSX already comes with the PixieDust library installed, but it is always a good idea to make sure you have the latest version
- In the first cell of the notebook enter: `!pip install --upgrade pixiedust`
- Click on the Run Cell (▸) button
- After the cell completes, if instructed to restart the kernel, from the notebook toolbar menu:
  - Go to > Kernel > Restart
  - Click Restart in the confirmation dialog

Note: The status of the kernel briefly flashes near the upper right corner, alerting when it is Not Connected, Restarting, Ready, etc.

Import PixieDust

- Before, you can use the PixieDust library it must be imported into the notebook
- In the next cell enter:

```
import pixiedust
```

- Click on the Run Cell (▶) button

Note: Whenever the kernel is restarted, the import pixiedust cell must be run before continuing.

PixieDust has been updated and imported, you are now ready to play with your data!

Import San Francisco traffic accidents data

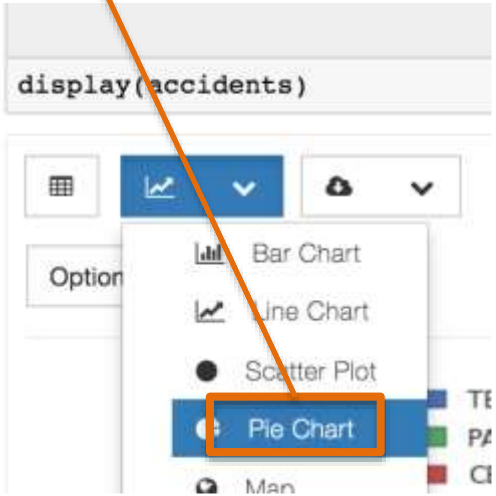
It would be a good idea to spend a few minutes browsing the San Francisco open data site: https://datasf.org/opendata/

- Load the data: With PixieDust, you can easily load CSV data from a URL into a PySpark DataFrame in the notebook.
- In a new cell, enter:

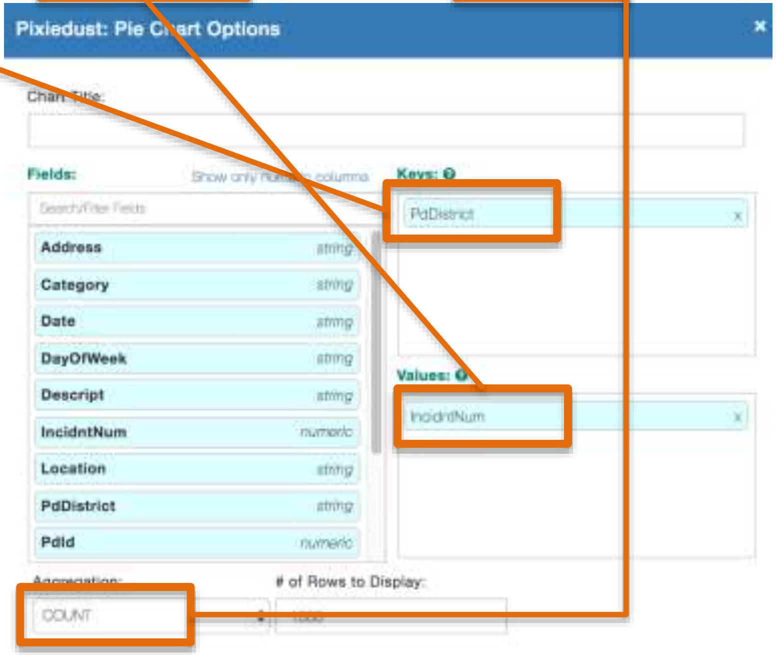
```
accidents =
pixiedust.sampleData("https://data.sfgov.org/api/views/vv57-
2fgy/rows.csv?accessType=DOWNLOAD")
```

Explore the Data

In the next few cells, we'll use the PixieDust `display()` to interactively explore the data. This tutorial will provide a few suggestions for which options to choose, but feel free to also explore on your own.
For example:
(Pie Chart - Options: Keys = PdDistrict, Values = IncidntNum, Aggregation = Count)



```
display(accidents)
```



Pixiedust: Pie Chart Options

Chart Title:

Fields: Show only numeric columns

Address	string
Category	string
Date	string
DayOfWeek	string
Descript	string
IncidntNum	numeric
Location	string
PdDistrict	string
PdId	numeric

Keys: PdDistrict

Values: IncidntNum

Aggregation: COUNT

# of Rows to Display: 1000

Initial Exploration

Let’s start exploring our data by answering the following questions

- In which police district do the most traffic accidents occur?

(Pie Chart - Options: Keys = PdDistrict, Values = IncidntNum, Aggregation = Count)

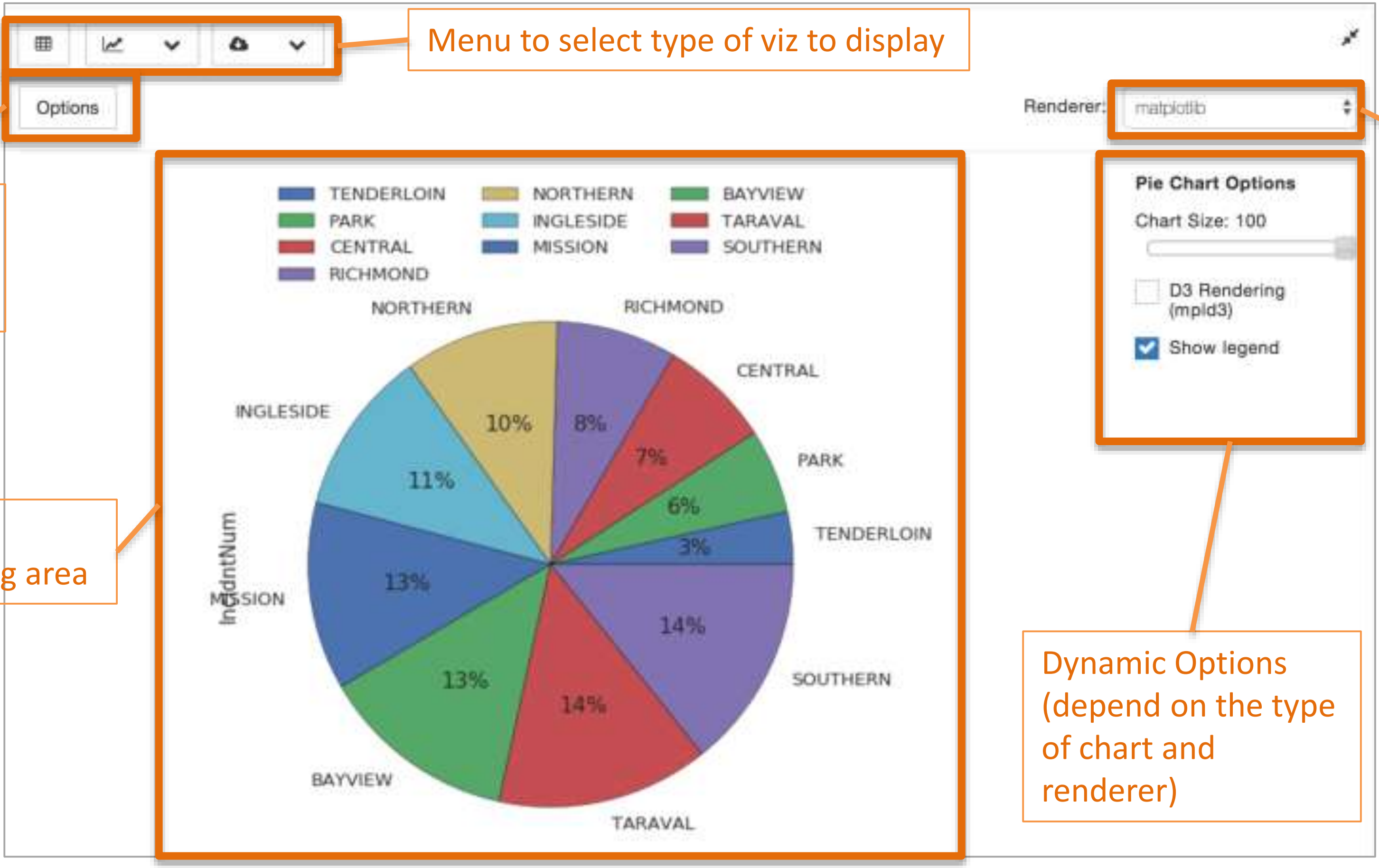
- We can also dig one level deeper by clustering by how each accident was resolved:

(Cluster By: Resolution)

- On what day of the week do the most traffic accidents occur?

(Bar Chart - Options: Keys = DayOfWeek, Values = IncidntNum, Aggregation = Count)

Results



Menu to select type of viz to display

Chart Options Dialog

Renderer Selector

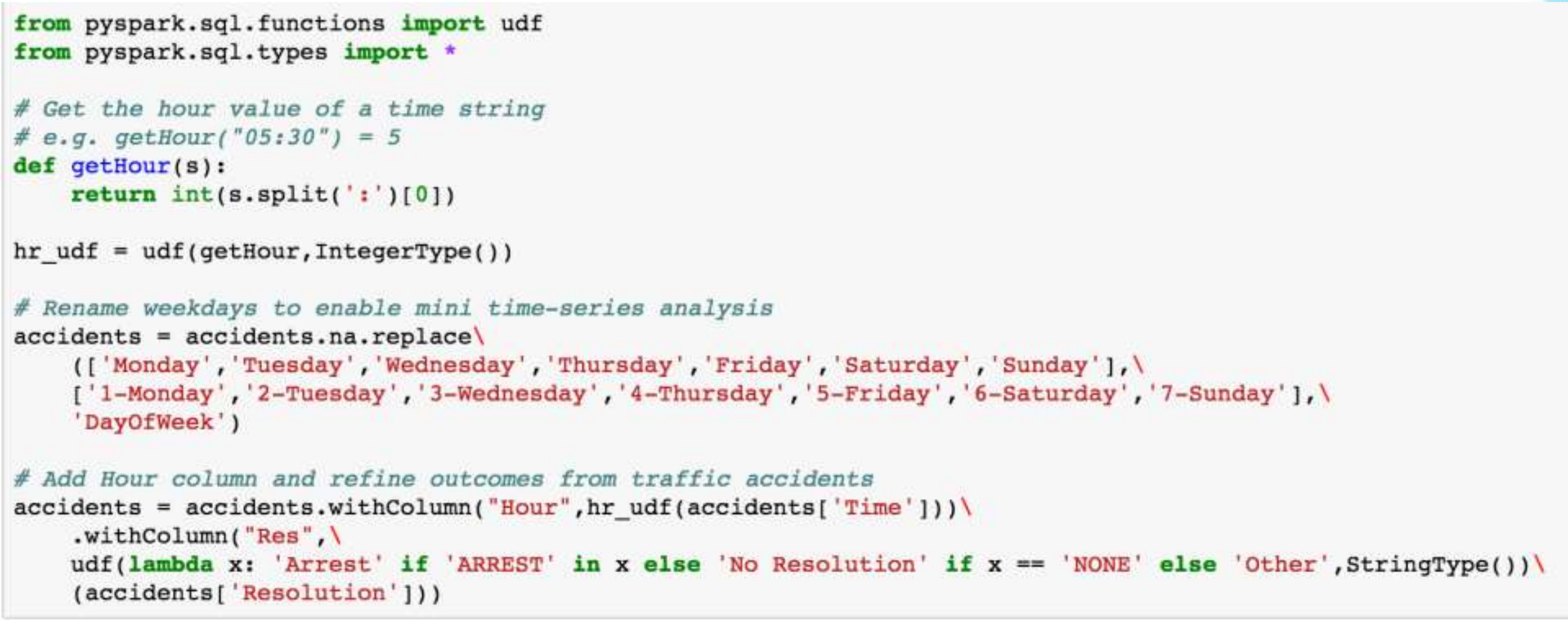
Chart Rendering area

Dynamic Options (depend on the type of chart and renderer)

More data exploration and hypothesis

- Immediately, we can identify a couple of areas of interest in our data without having to write a single line of code:
  1. Most accidents happen in the Southern and Taraval police districts
  2. Most accidents happen on Wednesdays and Thursdays.
- We can also see that our data needs some cleansing if we want to make analysis easier:
  1. The Time field needs to be converted into its time components
  2. Rename the DayOfWeek values so they are rendered in alphabetical order
- We should condense the outcome types of each traffic accident if we want to see the most common resolutions of traffic accidents in each police district, since the clustering above was unclear.

Cleansing the data



```python
from pyspark.sql.functions import udf
from pyspark.sql.types import *

# Get the hour value of a time string
# e.g. getHour("05:30") = 5
def getHour(s):
    return int(s.split(':')[0])

hr_udf = udf(getHour,IntegerType())

# Rename weekdays to enable mini time-series analysis
accidents = accidents.na.replace\
    (['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'],\
    ['1-Monday','2-Tuesday','3-Wednesday','4-Thursday','5-Friday','6-Saturday','7-Sunday'],\
    'DayOfWeek')

# Add Hour column and refine outcomes from traffic accidents
accidents = accidents.withColumn("Hour",hr_udf(accidents['Time']))\
    .withColumn("Res",\
    udf(lambda x: 'Arrest' if 'ARREST' in x else 'No Resolution' if x == 'NONE' else 'Other',StringType())\
    (accidents['Resolution']))
```

IBM

@DTAIEB55

More exploration on the cleansed data

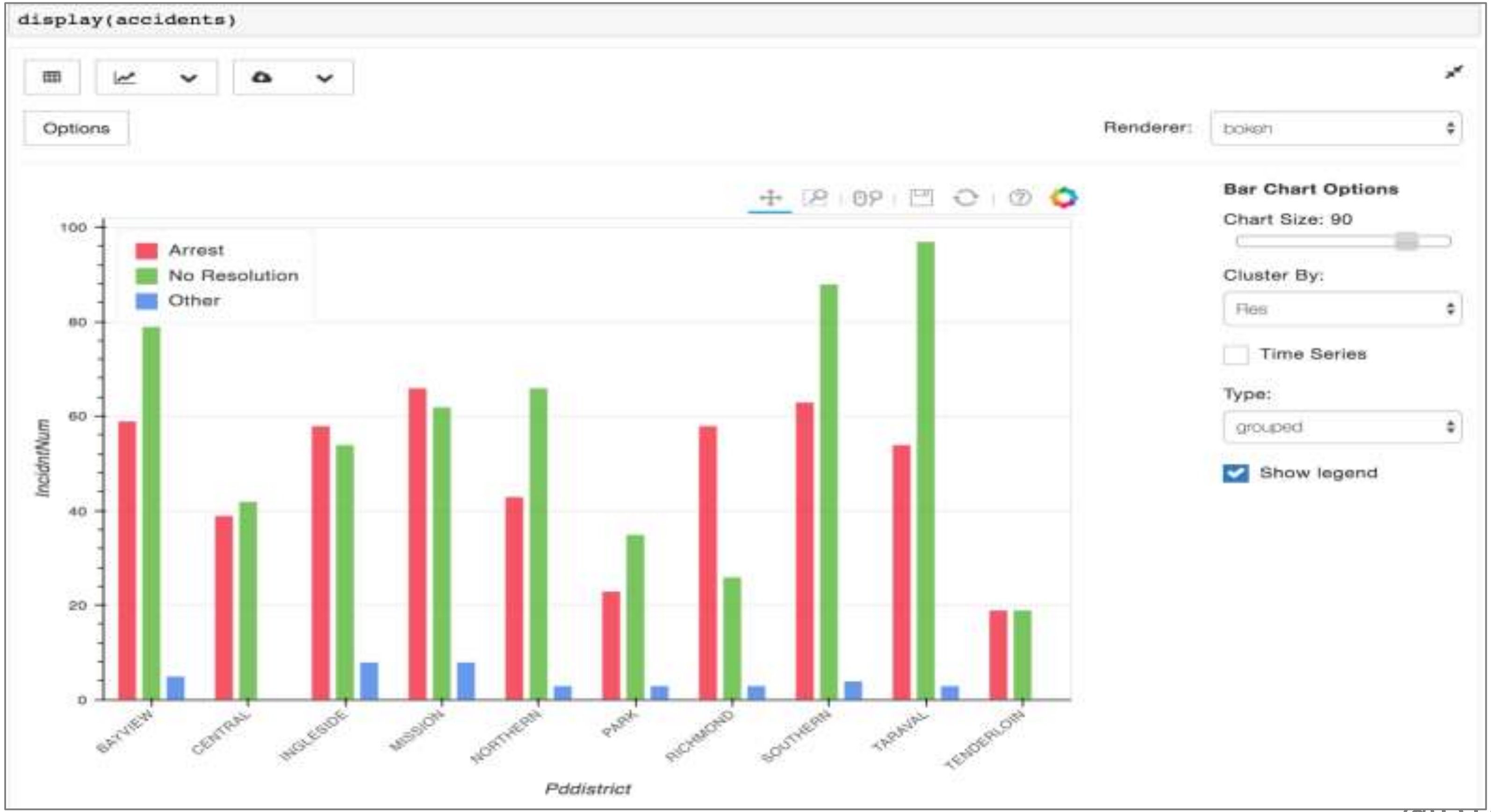
- Hypothesis: Do accidents in one police district result in more arrests than other police districts?

(Bar Chart - Options: Keys = PdDistrict, Values = IncidntNum, Aggregation = Count, Cluster By: Res)

- Question: How does the number of accidents change over the course of the week?

(Line Chart - Options: Keys = DayOfWeek, Values = IncidntNum, Aggregation = Count)

Some more results



```
display(accidents)
```

Options

Renderer: bokeh

Bar Chart Options

Chart Size: 90

Cluster By: Res

Time Series

Type: grouped

✓ Show legend

What have we learned

- A few lines of code makes it a lot easier to see that:
  1. Accidents in the Richmond police district are much more likely to result in arrest than all other districts
  2. The number of accidents peaks during the middle of the week, but decreases afterwards as the week winds down.

Now let's focus on the Taraval police district using some friendly SQL notation

Let’s use Spark SQL to focus on TARAVAL

- In a new cell, enter the following code to create a temporary SQL Table so we can then use regular a SQL query to select the accidents from TARAVAL.
- The results are then stored in a new Spark DataFrame

```
accidents.registerTempTable("accidents")
taraval = sqlContext.sql("SELECT * FROM accidents WHERE 
PdDistrict='TARAVAL'")
```

Exploration on TARAVAL accidents

- Question: Where in Taraval do most accidents happen?

(Map - Options: Keys = [X,Y], Values = IncidntNum, Aggregation = Count, Renderer: mapbox, kind: chloropleth-cluster)

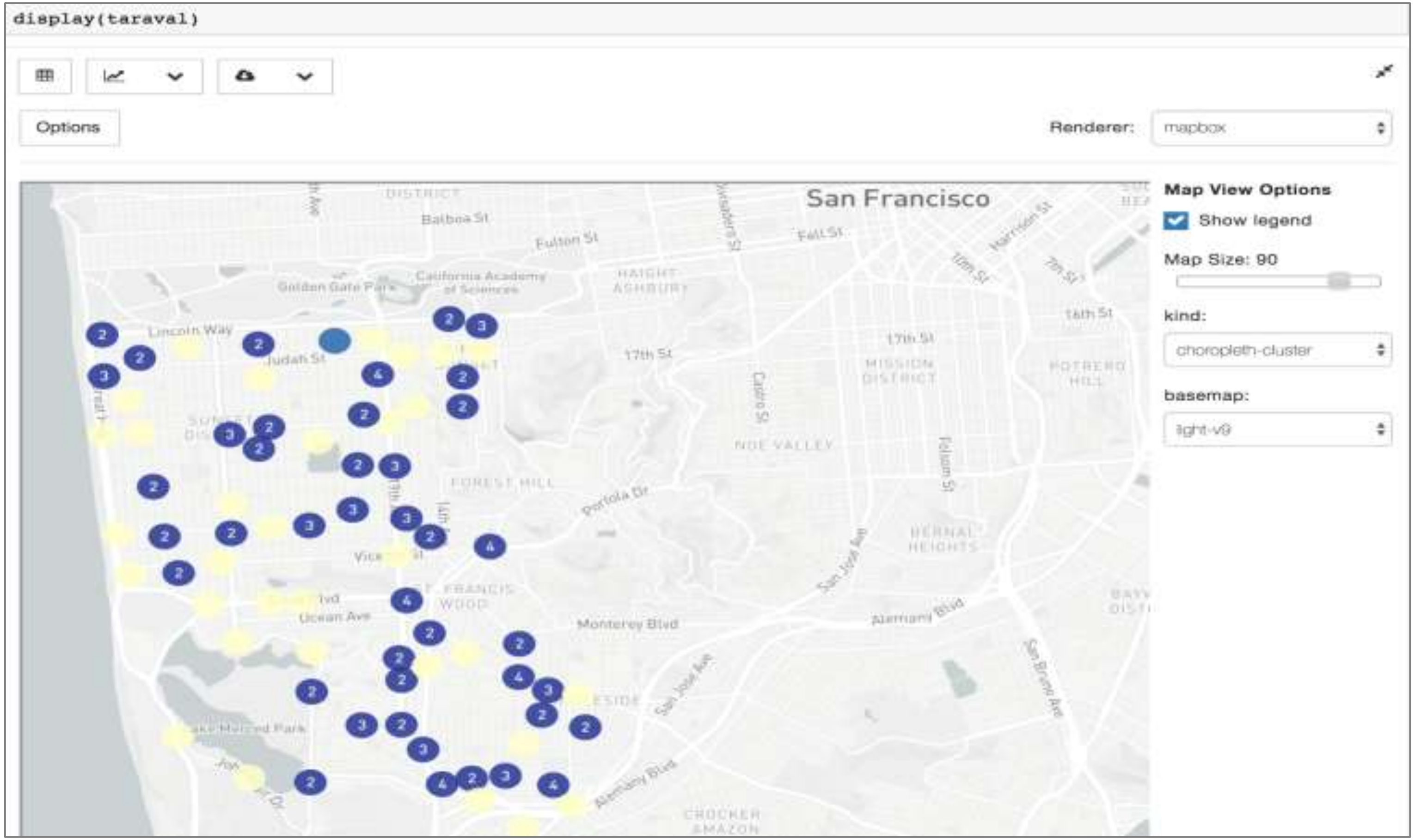
- Question: What time of day do most accidents occur?

(Line Chart - Options: Keys = Hour, Values = IncidntNum, Aggregation = Count)

Note: The Notebook comes with a public Mapbox access token. You can also create your own free Mapbox access token at the following url:

https://www.mapbox.com/help/create-api-access-token/

Map Visualization



```
display(taraval)
```

Options

Renderer: mapbox

Map View Options

✓ Show legend

Map Size: 90

kind: choropleth-cluster

basemap: light-v9

@DTAIEB55

Let’s recap

Most of the results from looking at the accident times are unsurprising:

- Less accidents during very early morning (people probably sleeping)
- Steady increase in number of accidents during morning commuting hours
- Less accidents during mid-evening (people probably eating dinner)
- (Sadly) more accidents late at night.

The interesting thing here is the sudden spike in accidents during mid-afternoon (2-3PM) - twice as many accidents happen during this two-hour window!

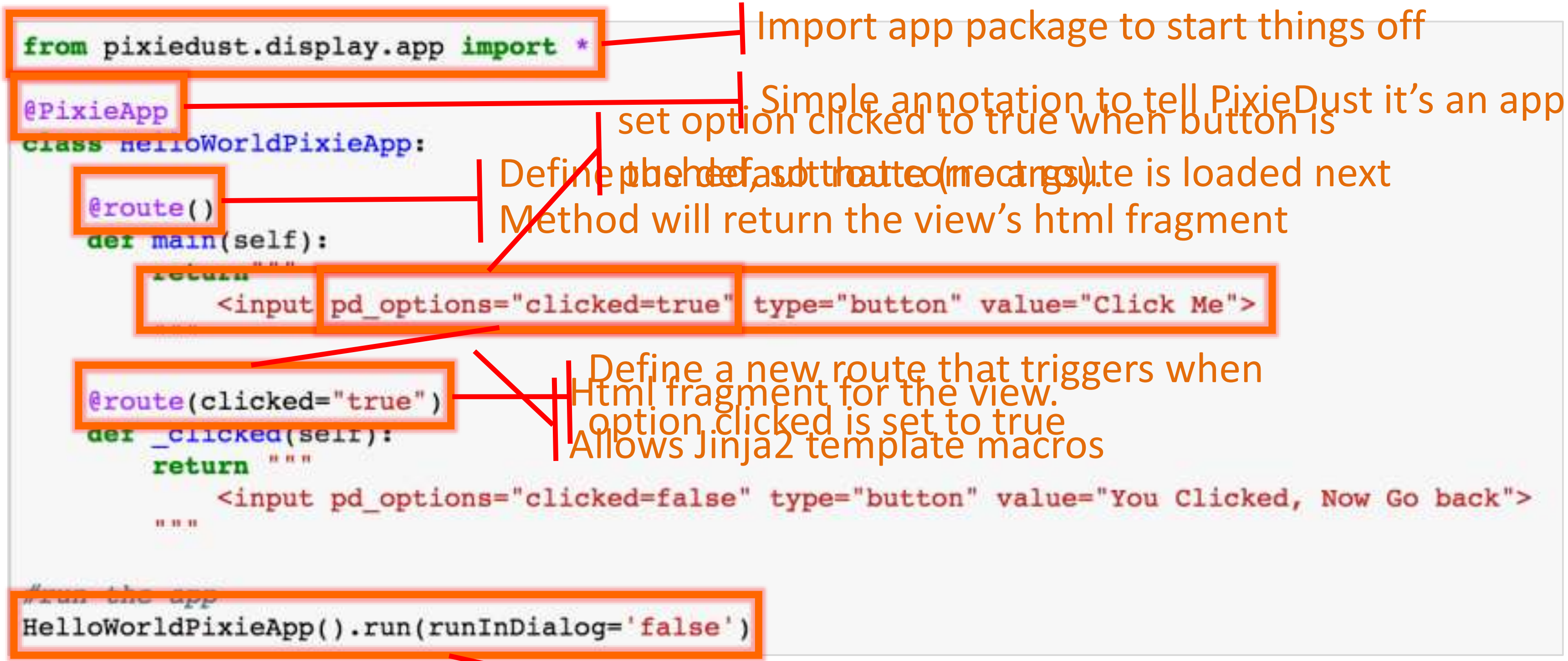
In the next section, we’ll bring more data to find more insights on the data. But instead of writing more code in linear cells, we’ll build a dashboard using another PixieDust feature called PixieApps

What are PixieApps

- PixieApps are Python classes that let you write UI for your analytics
- Easy to build: mostly HTML and CSS with some custom attributes
- Leverage PixieDust Display visualization for charting
- With PixieApps you can:
    - Create different html views with routes to invoke them
    - Invoke Python Scripts from user interactions
    - Run in the notebook cell output or in a Dialog
    - and much more…
- Use cases:
    - Dashboards
    - Data Browsers
    - Data Pipeline Management

Note: Please take the time to browse the PixieApp documentation page here:
https://ibm-cds-labs.github.io/pixiedust/pixieapps.html

PixieApp Hello Word



```
from pixiedust.display.app import *
@PixieApp
class HelloWorldPixieApp:
    @route()
    def main(self):
        return """
            <input pd_options="clicked=true" type="button" value="Click Me">
        """

    @route(clicked="true")
    def _clicked(self):
        return """
            <input pd_options="clicked=false" type="button" value="You Clicked, Now Go back">
        """

#run the app
HelloWorldPixieApp().run(runInDialog='false')
```

Import app package to start things off

Simple annotation to tell PixieDust it's an app

set option clicked to true when button is pushed, so that correct route is loaded next

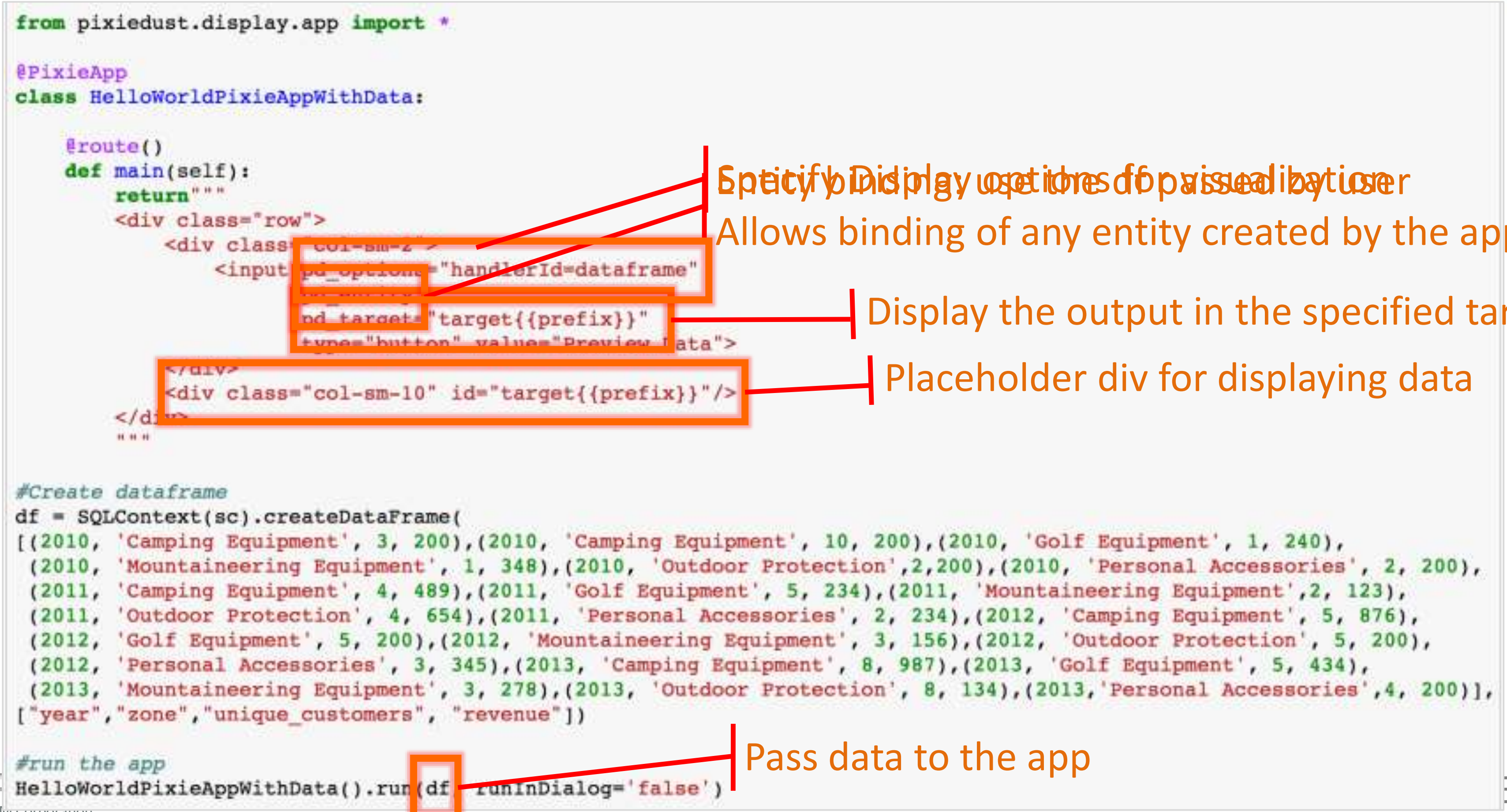
Define the default route (no args).
Method will return the view's html fragment

Define a new route that triggers when option clicked is set to true

Html fragment for the view.
Allows Jinja2 template macros

Import app package to start things off

PixieApp HelloWorld with Data



```
from pixiedust.display.app import *

@PixieApp
class HelloWorldPixieAppWithData:

    @route()
    def main(self):
        return"""
        <div class="row">
            <div class="col-sm-2">
                <input pd_options="handlerId=dataframe"
                pd_entity
                pd_target="target{{prefix}}"
                type="button" value="Preview Data">
            </div>
            <div class="col-sm-10" id="target{{prefix}}"/>
        </div>
        """

#Create dataframe
df = SQLContext(sc).createDataFrame(
[(2010, 'Camping Equipment', 3, 200),(2010, 'Camping Equipment', 10, 200),(2010, 'Golf Equipment', 1, 240),
 (2010, 'Mountaineering Equipment', 1, 348),(2010, 'Outdoor Protection',2,200),(2010, 'Personal Accessories', 2, 200),
 (2011, 'Camping Equipment', 4, 489),(2011, 'Golf Equipment', 5, 234),(2011, 'Mountaineering Equipment',2, 123),
 (2011, 'Outdoor Protection', 4, 654),(2011, 'Personal Accessories', 2, 234),(2012, 'Camping Equipment', 5, 876),
 (2012, 'Golf Equipment', 5, 200),(2012, 'Mountaineering Equipment', 3, 156),(2012, 'Outdoor Protection', 5, 200),
 (2012, 'Personal Accessories', 3, 345),(2013, 'Camping Equipment', 8, 987),(2013, 'Golf Equipment', 5, 434),
 (2013, 'Mountaineering Equipment', 3, 278),(2013, 'Outdoor Protection', 8, 134),(2013,'Personal Accessories',4, 200)],
["year","zone","unique_customers", "revenue"])

#run the app
HelloWorldPixieAppWithData().run(df, runInDialog='false')
```

Specify Display options for visualization

Entity binding: use the df passed by user

Allows binding of any entity created by the app

Display the output in the specified target

Placeholder div for displaying data

Pass data to the app

Back to our San Francisco Data

- In analyzing the geographical data, we can see a couple of clusters where accidents occur more frequently in Taraval - the southeastern corner looks particularly crowded.
- Some useful questions to ask at this point are:
  - Does crime has an effect on the number of accidents?
  - Are there more accidents in these areas because more people speed there
  - Do traffic calming devices reduce the number of accidents?

Note: In the next section, we'll download datasets from the San Francisco Open Data Web site: https://datasf.org/opendata

PixieApp Dashboard

Step 1: Create the skeleton

```
from pixiedust.display.app import *

@PixieApp
class SFDashboard():
    def mainScreen(self):
        return """
<div class="well">
    <center><span style="font-size:x-large">Analyzing San Francisco Public Safety
data with PixieDust</span></center>
    <center><span style="font-size:large"><a href="https://datasf.org/opendata"
target="new">https://datasf.org/opendata</a></span></center>
</div>
<div class="row">
    <div class="form-group col-sm-2" style="padding-right:10px;">
        <div><strong>Layers</strong></div>
</div>
    <div class="form-group col-sm-10">
        <div id="map{{prefix}}"/>
    </div>
</div>
"""
```

PixieApp Dashboard

Step 2: Create the map of accidents

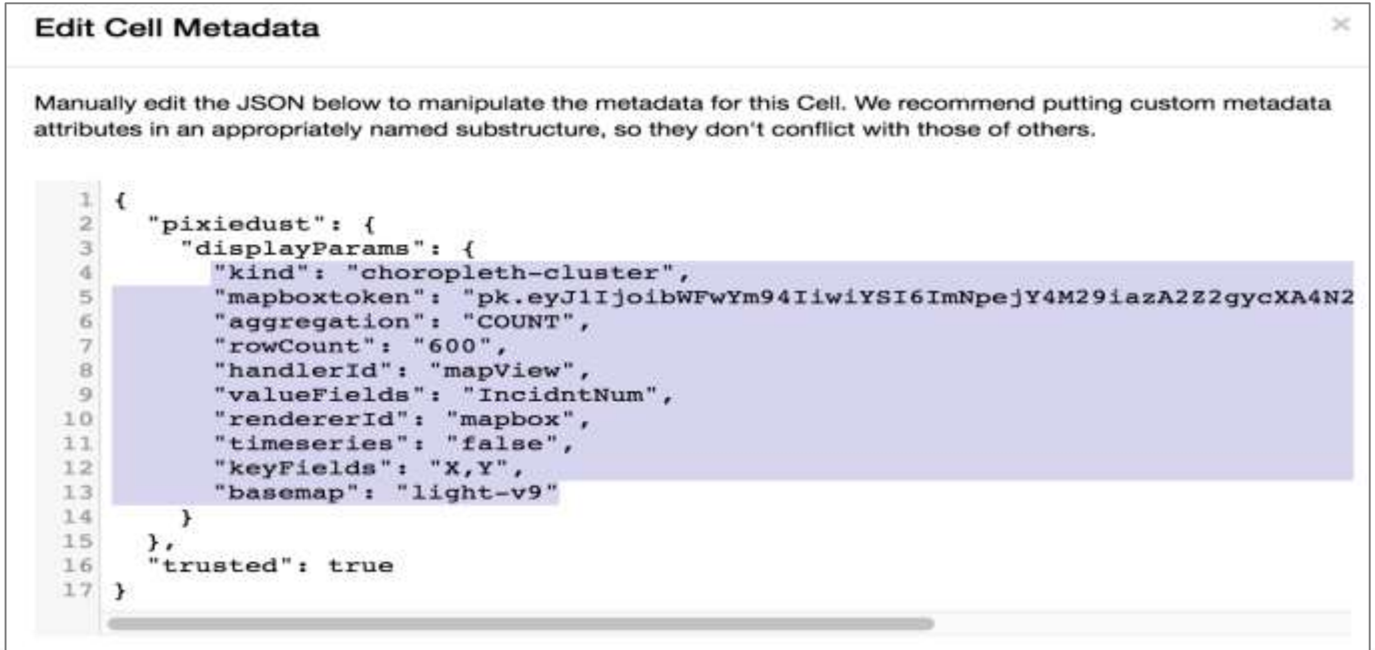
```
from pixiedust.display.app import *

@PixieApp
class SFDashboard():
    def mainScreen(self):
        return """
<div class="well">
    <center><span style="font-size:x-large">Analyzing San Francisco Public Safety data with
PixieDust</span></center>
    <center><span style="font-size:large"><a href="https://datasf.org/opendata"
target="new">https://datasf.org/opendata</a></span></center>
</div>
<div class="row">
    <div class="form-group col-sm-2" style="padding-right:10px;">
        <div><strong>Layers</strong></div>
</div>
    <div class="form-group col-sm-10">
        <div id="map{{prefix}}" pd_entity pd_options="{{this.formatOptions(this.mapJSONOptions)}}"/>
    </div>
</div>
"""
```

- What have we added:
  - pd_entity: tell PixieDust which dataset to work on
  - pd_options: Contains the PixieDust options for the map (see next slide for more info)

Generating the pd_options for the Map

The best way to generate the pd_options for a PixieDust visualization is to:

1. Call display() on a new cell
2. Graphically select the options for your chart
3. Select View/Cell Toobar/Edit metadata menu
4. Click on the “Edit Metadata” button and copy the pixiedust metadata



Edit Cell Metadata

Manually edit the JSON below to manipulate the metadata for this Cell. We recommend putting custom metadata attributes in an appropriately named substructure, so they don't conflict with those of others.

```
{
  "pixiedust": {
    "displayParams": {
      "kind": "choropleth-cluster",
      "mapboxtoken": "pk.eyJ1IjoibWFwYm94IiwiYSI6ImNpejY4M29iazA2Z2gycXA4N2
      "aggregation": "COUNT",
      "rowCount": "600",
      "handlerId": "mapView",
      "valueFields": "IncidntNum",
      "rendererId": "mapbox",
      "timeseries": "false",
      "keyFields": "X,Y",
      "basemap": "light-v9"
    }
  },
  "trusted": true
}
```

Format the PixieDust JSON Metadata

- To conform to the pd_options notation, we need to transform the PixieDust JSON metadata into an attribute string with the following format:

  “key1=value1;key2=value2;…”

- To make it easier, we use the a simple python transform function:

```
def formatOptions(self, options):
    return ';'.join(["{}={}".format(k,v) for (k, v) in iteritems(options)])
```

- The formatOptions is then invoked using JinJa2 notation from within the html

  pd_options = “{{this.formatOptions(this.mapJSONOptions)}}”

Initialize the pd_options

```
def setup(self):
    self.mapJSONOptions = {
      "mapboxtoken": ”XXXX",
      "chartsize": "90",
      "aggregation": "SUM",
      "rowCount": "500",
      "handlerId": "mapView",
      "rendererId": "mapbox",
      "valueFields": "IncidntNum",
      "keyFields": "X,Y",
      "basemap": "light-v9"
    }
```

Note: setup is a special method that will be called automatically when the PixieApp is initialized.

PixieApp Dashboard

Step 3: Create the GeoJSON customer Layers

```
from pixiedust.display.app import *
from pixiedust.apps.mapboxBase import MapboxBase

@PixieApp
class SFDashboard(MapboxBase):
    def setup(self):
      …
      self.setLayers([
          {
              "name": "Speeding",
              "url": "https://data.sfgov.org/api/geospatial/mfjz-
pnye?method=export&format=GeoJSON"
          },
          {
              "name": "Traffic calming",
              "url": "https://data.sfgov.org/api/geospatial/ddye-
rism?method=export&format=GeoJSON",
              "type": "symbol",
              "layout": {
                  "icon-image": "police-15",
                  "icon-size": 1.5
              }
          },
          …
          ])
```

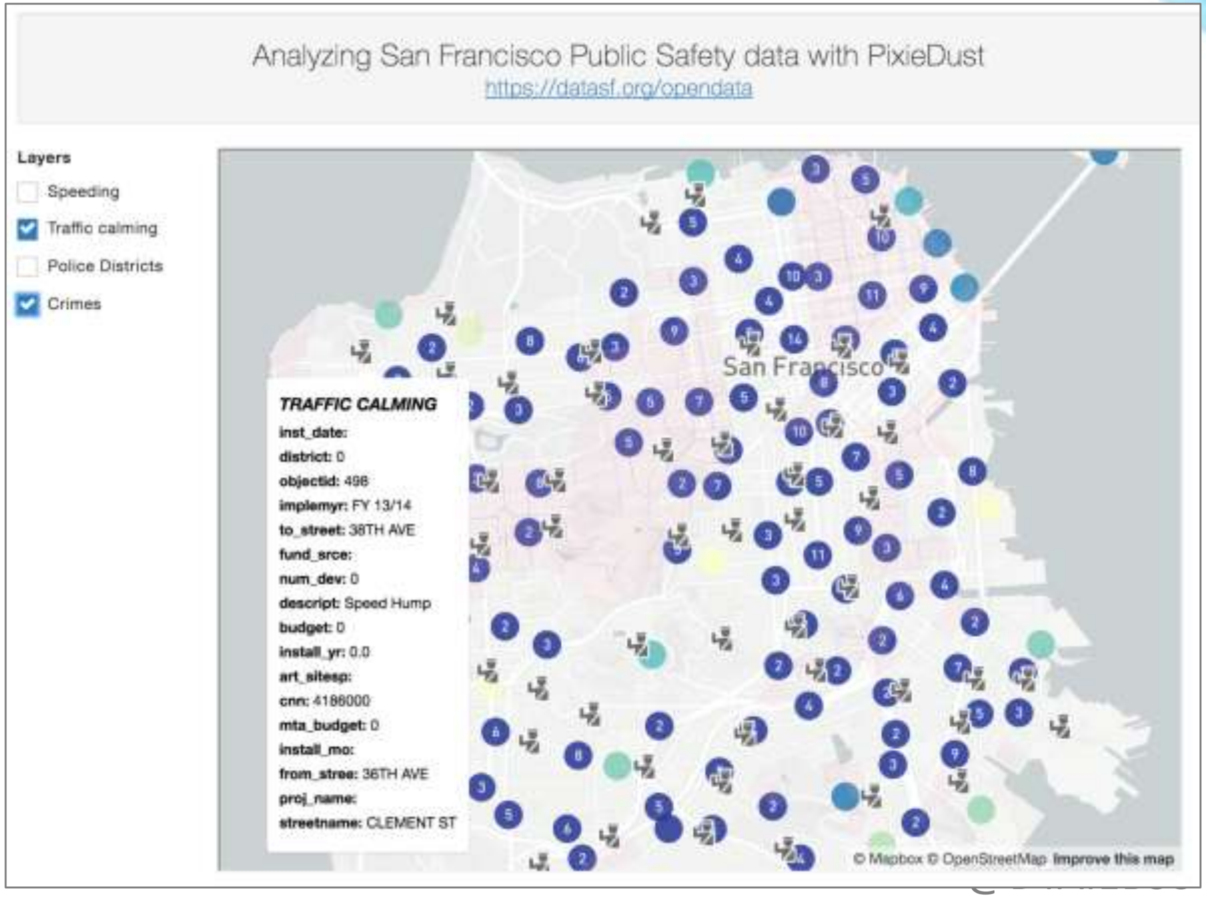
PixieApp Dashboard

Step 4: Create the checkboxes from the layers

```
from pixiedust.display.app import *
from pixiedust.apps.mapboxBase import MapboxBase

@PixieApp
class SFDashboard(MapboxBase):
    …
    @route()
    def mainScreen(self):
        return """
<div class="well">
    <center><span style="font-size:x-large">Analyzing San Francisco Public Safety data with 
PixieDust</span></center>
    <center><span style="font-size:large"><a href="https://datasf.org/opendata" 
target="new">https://datasf.org/opendata</a></span></center>
</div>
<div class="row">
    <div class="form-group col-sm-2" style="padding-right:10px;">
        <div><strong>Layers</strong></div>
        {% for layer in this.layers %}
        <div class="rendererOpt checkbox checkbox-primary">
            <input type="checkbox" pd_refresh="map{{prefix}}" pd_script="self.toggleLayer({{loop.index0}})">
            <label>{{layer["name"]}}</label>
        </div>
        {%endfor%}
    </div>
    <div class="form-group col-sm-10">
        <div id="map{{prefix}}" pd_entity pd_options="{{this.formatOptions(this.mapJSONOptions)}}"/>
    </div>
</div>
"""
```

Run the PixieApp Dashboard

You can find the final code for the PixieApp in the complete notebook :http://ibm.biz/PixieDustBootcampSFNotebook


Analyzing San Francisco Public Safety data with PixieDust
https://datasf.org/opendata

Layers
- [ ] Speeding
- [✓] Traffic calming
- [ ] Police Districts
- [✓] Crimes

TRAFFIC CALMING
inst_date:
district: 0
objectid: 498
implemyr: FY 13/14
to_street: 38TH AVE
fund_srce:
num_dev: 0
descript: Speed Hump
budget: 0
install_yr: 0.0
art_sitesp:
cnn: 4186000
mta_budget: 0
install_mo:
from_stree: 36TH AVE
proj_name:
streetname: CLEMENT ST

Final Thoughts

- In this tutorial:
  - We showed how PixieDust help notebook users (data scientists or not) to load and visualize data without having to write code
  - We also showed how to use notebooks in new ways thanks to PixieApps by writing dashboard
- Next Steps
  - Improve the analytics by finding new insights on the SF Traffic accidents dataset, or perhaps start a new notebook with a new dataset from https://datasf.org/opendata/
  - Improve the PixieApp dashboard by adding new layers. Just find the geojson link from the opendata site by clicking on the explore button (warning: not all datasets come with a geojson link). Then just add a new entry in the layers array