How to Build a Reporting Dashboard using Dash and Plotly

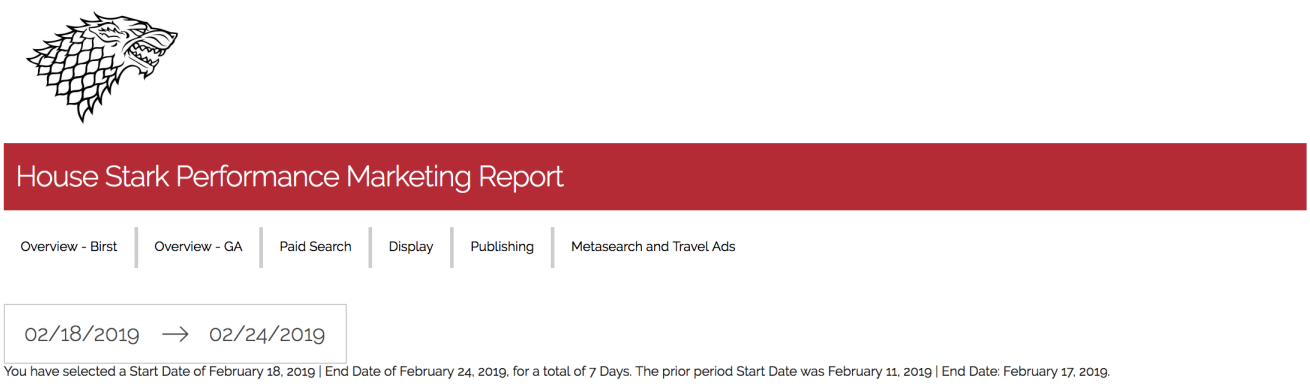
David Comfort
Mar 11, 2019 · 24 min read ★

In this blog post, I will provide a step-by-step tutorial on how to build a reporting dashboard using Dash, a Python framework for building analytical web applications. Rather than go over the basics of building a Dash app, I provide a detailed guide to building a multi-page dashboard with data tables and graphs.

I built the reporting dashboard as a multi-page app in order to break up the dashboard into different pages so it less overwhelming and to present data in an organized fashion. On each dashboard page, there are two data tables, a date range selector, a data download link, as well as a set of graphs below the two dashboards. I ran into several technical challenges while building the dashboard and I describe in detail how I overcame these challenges.

The completed Dashboard can be viewed at https://davidcomfort-dash-app1.herokuapp.com/cc-travel-report/paid-search/ and the code is provided in Github. (The data used in the app is random data and the names for the products are “dummy” names.)

. . .



Paid Search

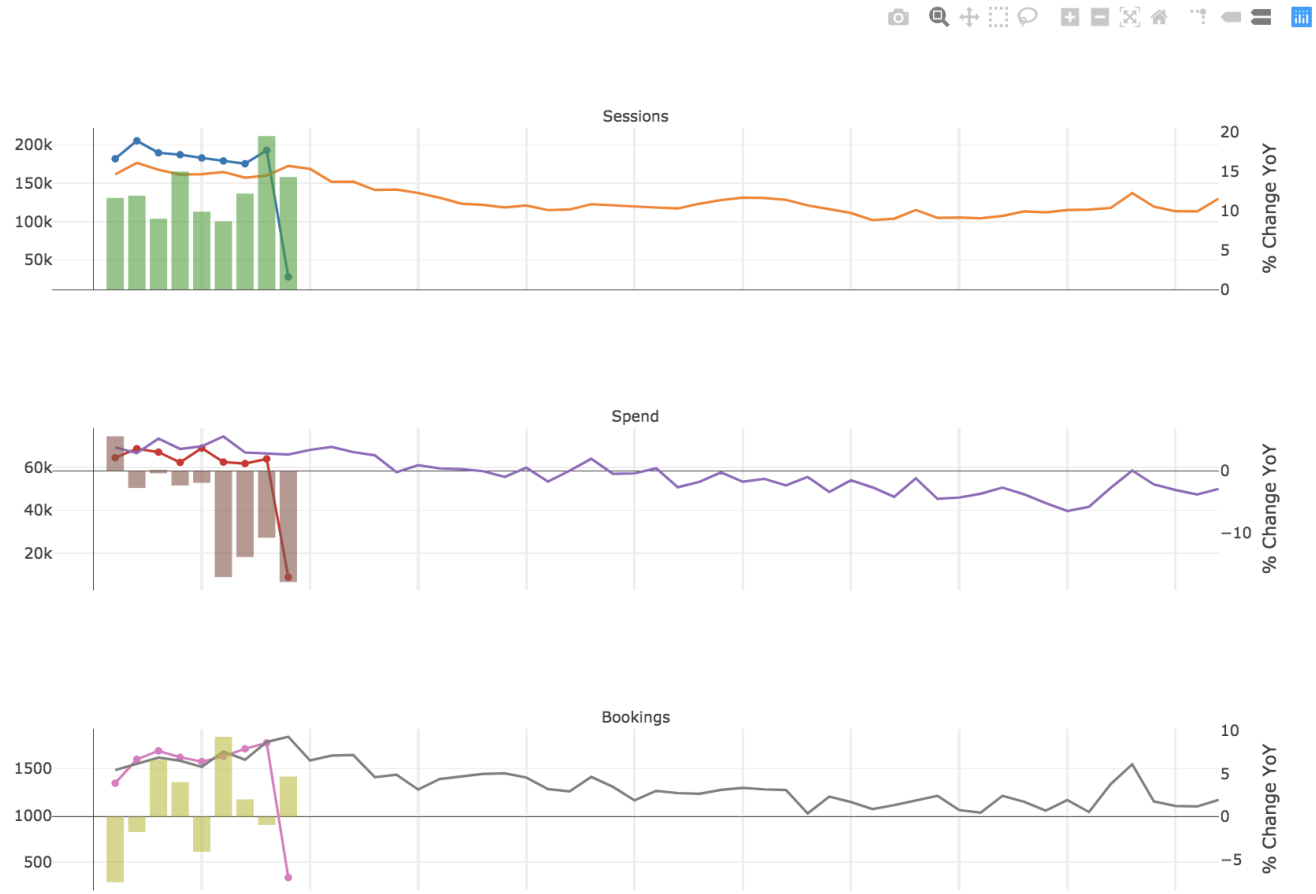
Condensed Data Table Complete Data Table

Placement type	Spend TY	Spend PoP (%)	Spend YoY (%)	Sessions - TY	Sessions PoP (%)	Sessions YoY (%)	Bookings - TY	Bookings PoP (%)	Bookings YoY (%)
Paid Search Brand	$61,882	(2.0%)	(13.3%)	192,118	7.4%	18.3%	1,828	5.0%	(2.0%)
Paid Search Non-Brand Prospecting DSA	$89,166	(16.6%)	35.6%	66,401	4.9%	28.8%	612	(7.6%)	(2.9%)
Paid Search Non-Brand Prospecting Destination	$187,833	4.2%	26.9%	137,642	10.5%	51.5%	625	(6.4%)	(18.0%)
Paid Search Non-Brand Prospecting Generics	$71,578	(12.6%)	8.7%	68,308	4.1%	11.2%	651	13.8%	(10.7%)
Paid Search Non-Brand Prospecting Property	$44,909	(0.4%)	(38.7%)	42,944	15.5%	(23.7%)	605	(4.0%)	(11.7%)
Paid Search Non-Brand Retargeting DSA	$62,141	46.6%	(33.6%)	33,578	2.9%	(26.0%)	621	2.0%	(19.6%)
Paid Search Non-Brand Retargeting Destination	$165,738	(1.4%)	452.0%	87,549	11.1%	21.2%	838	3.8%	0.0%
Paid Search Non-Brand Retargeting Generics	$92,577	(6.8%)	(59.7%)	53,169	4.0%	(1.1%)	598	(9.9%)	(21.8%)
Paid Search Non-Brand Retargeting Property	$69,056	2.7%	195.3%	47,853	15.5%	3.0%	633	0.6%	(3.8%)

DOWNLOAD DATA

Placement type	CPS - TY	CPS - LP	CPS PoP (Abs)	CPS PoP (%)	CPS - LY	CPS YoY (Abs)	CPS YoY (%)	CVR - TY	CVR - LP	CVR PoP (Abs)	CVR PoP (%)	CVR - LY	CVR YoY (Abs)
Paid Search Brand	$0.32	$0.35	($0.03)	(8.8%)	$0.44	($0.12)	(26.7%)	0.95%	0.97%	(0.02%)	(2.3%)	1.15%	(0.20%)
Paid Search Non-Brand Prospecting DSA	$1.34	$1.69	($0.35)	(20.5%)	$1.27	$0.07	5.3%	0.92%	1.05%	(0.12%)	(11.9%)	1.22%	(0.30%)
Paid Search Non-Brand Prospecting Destination	$1.36	$1.45	($0.08)	(5.7%)	$1.63	($0.27)	(16.3%)	0.45%	0.54%	(0.08%)	(15.3%)	0.84%	(0.38%)
Paid Search Non-Brand Prospecting Generics	$1.05	$1.25	($0.20)	(16.1%)	$1.07	($0.02)	(2.3%)	0.95%	0.87%	0.08%	9.3%	1.19%	(0.23%)
Paid Search Non-Brand Prospecting Property	$1.05	$1.21	($0.17)	(13.8%)	$1.30	($0.26)	(19.7%)	1.41%	1.69%	(0.29%)	(16.9%)	1.22%	0.19%
Paid Search Non-Brand Retargeting DSA	$1.85	$1.30	$0.55	42.5%	$2.06	($0.21)	(10.3%)	1.85%	1.87%	(0.02%)	(0.9%)	1.70%	0.15%
Paid Search Non-Brand Retargeting Destination	$1.89	$2.13	($0.24)	(11.2%)	$0.42	$1.48	355.4%	0.96%	1.02%	(0.07%)	(6.5%)	1.16%	(0.20%)
Paid Search Non-Brand Retargeting Generics	$1.74	$1.94	($0.20)	(10.3%)	$4.27	($2.53)	(59.2%)	1.12%	1.30%	(0.17%)	(13.4%)	1.42%	(0.30%)
Paid Search Non-Brand Retargeting Property	$1.44	$1.62	($0.18)	(11.1%)	$0.50	$0.94	186.6%	1.32%	1.52%	(0.20%)	(12.9%)	1.42%	(0.09%)

Figure 1: Dashboard built using Dash



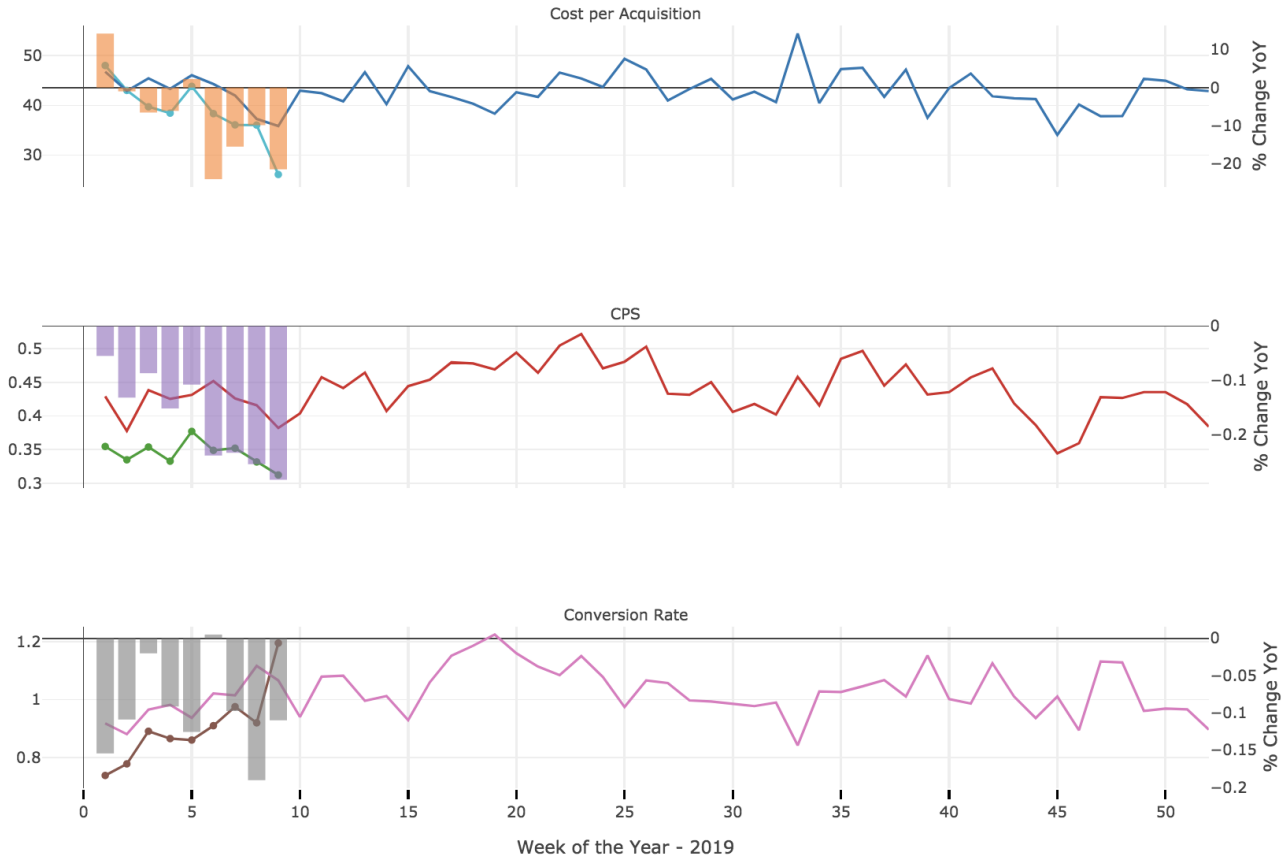


Figure 2: Second Part of Dashboard

Table of Contents

. . .

1. Introduction

In January of this year, I went to my first PyData conference, PyData Miami, and sat in on a great presentation by Chelsea Douglas about Dash.

Embedded Video 1: Dash: data exploration web apps in pure Python — Chelsea Douglas

It quickly became apparent how powerful Dash was and how I easily I could build web apps and dashboards using Python.

From my perspective, there was a real need in my company to automate reporting, replace Microsoft Excel pivot tables and spreadsheets, and provide an alternative to Business Intelligence tools. Even though various stakeholders in my company have relied upon Excel spreadsheets for regular reporting, their usage becomes unwieldy, they are prone to error, they are not platform independent, and they do not lend themselves to automation.

Therefore, I endeavored to build a multi-page web application using Dash. This article goes into the nitty, gritty details of my efforts and how I overcame several technical challenges. I should note that I come from a data scientist perspective and make no claims to be a software developer. Hence, there certainly is room for improvement in my code and I would welcome the reader's suggestions. At the same time, I hope the reader can benefit from my efforts if they need to build complex dashboards and data tables.

. . .

2. What am I Trying to Achieve with a Dashboard?

At my present company, a lot of periodic reporting is done either with Excel spreadsheets and pivot tables, or using business intelligence tools such as Birst or Qlikview. Hence, I wanted to build a reporting dashboard as a proof-of-concept that could replace and enhance our reporting. Specifically, I wanted to build a reporting web application for one of brands, which could report out on different marketing channel metrics, enable automation and provide ease of access.

The requirements for the **E-commerce Marketing Dashboard** included:

- Break out the different marketing channels into different pages so that the amount of data presented in the dashboard would not be overwhelming.
- A date selector so a user can select a date range to filter the data.
- A data table to present basic metrics (spend, sessions, number of transactions, and revenue) for each digital marketing channel or product type for the date range selected, as well as for the same period last year and a corresponding period prior to the date range selected.*
- Another data table to present calculated metrics for the date range selected (as well as for the same period last year and prior period). These metrics include cost-per-session (CPS), conversion rate (CVR), and CPA (cost-per-acquisition).
- A link to download the data (in Excel format) that is displayed in each data table.
- Graphs of the metrics on a weekly basis are to be depicted below the data tables.
- *An example of a corresponding prior period would be if a user selected week numbers 5 and 6 of 2019, then the corresponding prior period would be week numbers 3 and 4 of 2019.*

. . .

3. Challenges Building the Dashboard

There were multiple challenges that cropped up when I was building the dashboard app. Some of the main challenges included:

- Figuring out a method to display either a condensed data table or a complete data table.
- Having the ability to color code cells in the data tables, based upon the value in that cell.
- Having the ability to select multiple products to include in a graphical representation of the data.
- Having the ability to update the metrics in the data tables on-the-fly, depending upon the date range a user selects.
- Having the ability to download the data presented in the data tables without any added formatting.
- Depicting year-to-year changes in metrics on the same x-axis as a given metric.
- Being able to zoom in on all graphs at the same time.

. . .

4. Creating a New Environment and Installing Dash

There is a Dash User Guide, which provides a fairly thorough introduction to Dash and I encourage the reader to go through the user guide and build some simple Dash apps prior to tackling a full fledged dashboard. In addition, there is a Dash Community Forum, a show-and-tell section of the forum highlighting work by the Dash community, a gallery of Dash projects, a curated list of awesome Dash resources, and an introductory essay about Dash:

> *Dash is a Open Source Python library for creating reactive, Web-based applications. Dash started as a public proof-of-concept on GitHub 2 years ago. We kept this prototype online, but subsequent work on Dash occurred behind closed doors. We used feedback from private trials at banks, labs, and data science teams to guide the product forward.* ***Today, we're excited to announce the first public release of Dash that is both enterprise-ready and a first-class member of Plotly's open-source tools.*** *Dash can be downloaded today*

> *from Python's package manager with `pip install dash` — it's entirely open-source and MIT licensed. You'll find a getting started guide here and the Dash code on GitHub here.*
>
> *Dash is a user interface library for creating analytical web applications. Those who use Python for data analysis, data exploration, visualization, modelling, instrument control, and reporting will find immediate use for Dash.*
>
> *Dash makes it dead-simple to build a GUI around your data analysis code.*

Prior to installing Dash, as is the usual practice, I created a new environment using conda: `conda create --name dash` and then activated that environment, `conda activate dash` . I then simply followed the Dash installation protocol provided in the user guide:

```
pip install dash==0.39.0  # The core dash backend
pip install dash-html-components==0.14.0  # HTML components
pip install dash-core-components==0.44.0  # Supercharged components
pip install dash-table==3.6.0  # Interactive DataTable component (new!)
pip install dash-daq==0.1.0  # DAQ components (newly open-sourced!)
```

dashboard_pip_install_dash hosted with ❤ by **GitHub** **view raw**

Code Block 1: Pip installing Dash and its components

I should note that the versions for Dash and its components will change from above and you should refer to the User Guide.

. . .

5. Getting Started with Dash

There are already quite a few tutorials for Dash, so I will focus on how to build a multi-page dashboard with data tables and graphs in this tutorial, rather than go over the basics of building a Dash app.

If you are just getting started in Dash, I would encourage the reader to go through at least the first three sections of the excellent Dash User Guide. There is also a section on the Dash Data Table. There are several tutorials to get you started*:

- Introducing Plotly Dash — A high level introduction to Dash by Chris Parmer, the author of Dash. This essay was released as part of Dash's official launch (June 21, 2017).

- Plotly's tutorials — Part 1: App Layout
- Plotly's tutorials — Part 2: Interactivity
- Plotly's tutorials — Part 3: Interactive Graphing
- Plotly's tutorials — Part 4: Callbacks With State
- Interactive Web-Based Dashboards in Python — How the MVC model pertains to Dash and a walkthrough of building an app.
- Using Plotly's Dash to deliver public sector decision support dashboards — Building a complex dashboard step-by-step.
- OPS CodeDay: Dash Plotly Map + Graph — How to use Jupyter notebooks in tandem with Dash to create mapping viz.
- Creating Interactive Visualizations with Plotly's Dash Framework — High level overview of how to get started with Dash.
- Finding Bigfoot with Dash, Part 1 — Walkthrough of building a dashboard of Bigfoot sightings. Part 2, Part 3.
- Visualize Earthquakes with Plotly Dash — Environmental scan of alternatives to Dash followed with a tutorial.
- ARGO Labs — Plotly Dash Tutorial (Video) — Detailed introduction to creating interactive dashboards.
- Data Visualization GUIs with Dash and Python (Video playlist) — Five-part series exploring Dash features.

**From the Awesome Dash Github page.*

Essentially, Dash apps are composed of two parts: (1) the "*layout*" of the app that describes the look and feel of the app, and (2) the "*callbacks*" that enable the apps to be interactive. A simple Dash App Layout is presented in the user guide and reproduced below:

```
1  import dash
2  import dash_core_components as dcc
3  import dash_html_components as html
4  from dash.dependencies import Input, Output
5
```

```
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.Input(id='my-id', value='initial value', type='text'),
    html.Div(id='my-div')
])


@app.callback(
    Output(component_id='my-div', component_property='children'),
    [Input(component_id='my-id', component_property='value')]
)
def update_output_div(input_value):
    return 'You\'ve entered "{}"'.format(input_value)


if __name__ == '__main__':
    app.run_server(debug=True)
```

dashboard_simple_dash_app.py hosted with ❤ by **GitHub** **view raw**

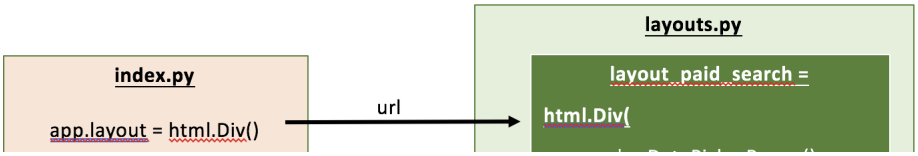
Code Block 2: Simple Dash App Layout

The dashboard which I describe in this tutorial splits up the Dash app into different files and enables one to build a multi-page app.

There is an “index” page which renders different pages, or layouts, depending the URL. Each individual layout consists of several different Dash components, including a date range picker, data tables, a download link, and several graphs. Each of these components is related to one or more “callbacks” that enable the dashboard to be interactive.

A user can interact with one of the Dash components (e.g., change the date range) and the other components reflect this change (e.g., cause the data presented in the data tables to change).

A schematic of this approach is in the following diagram:



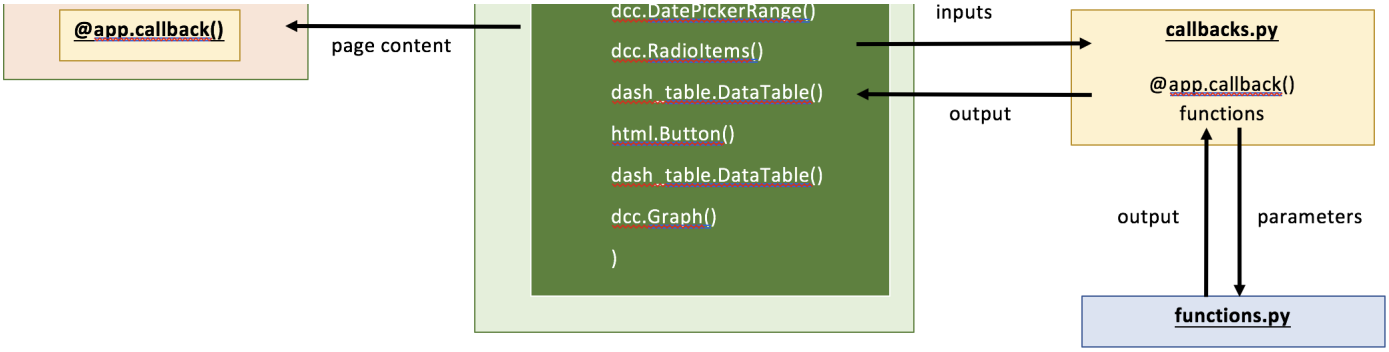


Figure 3: Schematic of Dashboard files

. . .

6. Building a Multi-page Application

The structure of multi-page Dash application, based upon the Dash documentation at https://dash.plot.ly/urls, with slight tweaks, is presented below:

```
- __init__.py
- app.py
- assets
    |-- logo.jpeg
    |-- custom.css
    |-- favicon.ico
- callbacks.py
- components
    |-- __init__.py
    |-- functions.py
    |-- header.py
    |-- printButton.py
- data
    |-- datafile.csv
- index.py
- layouts.py
- Procfile
- requirements.txt
```

dashboard_structure.txt hosted with ❤ by **GitHub** **view raw**

Code Block 3: File Structure of a Multi-page Dash App

The file `app.py` simply has the following:

```
import dash
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
```

```
app = dash.Dash(__name__, external_stylesheets=external_stylesheets, url_base_pathname=
server = app.server
app.config.suppress_callback_exceptions = True
import dash_auth
# Keep this out of source code repository - save in a file or a database
VALID_USERNAME_PASSWORD_PAIRS = [
    ['[username]', '[password']
]
auth = dash_auth.BasicAuth(
    app,
    VALID_USERNAME_PASSWORD_PAIRS
)
```

dashboard_app.py hosted with ❤ by **GitHub** **view raw**

Code Block 4: app.py

Note that I am using the dash authentication module (https://dash.plot.ly/authentication). This has some implications which I will go into below.

Building the Index Page

The `index.py` file essentially defines the URL page structure of the app by defining a function `display_page`, which determines what page layout should be rendered depending upon the app URL.

```
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

# see https://community.plot.ly/t/nolayoutexception-on-deployment-of-multi-page-dash-ap
from app import server
from app import app
from layouts import layout_birst_category, layout_ga_category, layout_paid_search, noPa
import callbacks

# see https://dash.plot.ly/external-resources to alter header, footer and favicon
app.index_string = '''
<!DOCTYPE html>
<html>
    <head>
        {%metas%}
        <title>CC Performance Marketing Report</title>
        {%favicon%}
        {%css%}
```

```
    </head>
    <body>
        {%app_entry%}
        <footer>
            {%config%}
            {%scripts%}
        </footer>
        <div>CC Performance Marketing Report</div>
    </body>
</html>
'''

app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])

# Update page
# # # # # # # # #
@app.callback(Output('page-content', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):
    if pathname == '/cc-travel-report' or pathname == '/cc-travel-report/overview-birst
        return layout_birst_category
    elif pathname == '/cc-travel-report/overview-ga/':
        return layout_ga_category
    elif pathname == '/cc-travel-report/paid-search/':
        return layout_paid_search
    elif pathname == '/cc-travel-report/display/':
        return layout_display
    elif pathname == '/cc-travel-report/publishing/':
        return layout_publishing
    elif pathname == '/cc-travel-report/metasearch-and-travel-ads/':
        return layout_metasearch
    else:
        return noPage

# # # # # # # # #
external_css = ["https://cdnjs.cloudflare.com/ajax/libs/normalize/7.0.0/normalize.min.c
                "https://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.min.css
                "//fonts.googleapis.com/css?family=Raleway:400,300,600",
                "https://codepen.io/bcd/pen/KQrXdb.css",
                "https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.mi
                "https://codepen.io/dmcomfort/pen/JzdzEZ.css"]

for css in external_css:
    app.css.append_css({"external_url": css})

```

```
external_js = ["https://code.jquery.com/jquery-3.2.1.min.js",
               "https://codepen.io/bcd/pen/YaXojL.js"]

for js in external_js:
    app.scripts.append_script({"external_url": js})

if __name__ == '__main__':
    app.run_server(debug=True)
```

This is very similar to the `index.py` page in the Dash Vanguard Report at https://github.com/plotly/dash-vanguard-report, with a demo at (https://dash-gallery.plotly.host/dash-vanguard-report/portfolio-management).

However, I had to include the line `from app import server`, in order to overcome an issue when I deployed the app on Heroku (see https://community.plot.ly/t/nolayoutexception-on-deployment-of-multi-page-dash-app-example-code/12463 for details).

Customizing the Page Title and Favicon

In addition, I wanted to customize the app's HTML index template, specifically, the page title and the favicon. In order to do so, I added the following `index_string:` to `index.py`:

```
# see https://dash.plot.ly/external-resources to alter header, footer and favicon
app.index_string = '''
<!DOCTYPE html>
<html>
    <head>
        {%metas%}
        <title>CC Performance Marketing Report</title>
        {%favicon%}
        {%css%}
    </head>
    <body>
        {%app_entry%}
        <footer>
            {%config%}
            {%scripts%}
        </footer>
        <div>CC Performance Marketing Report</div>
    </body>
```

```
</html>
'''
```

dashboard_app_index_string.py hosted with ❤ by GitHub view raw

Code Block 6: index_string in index.py File

In order to order to customize the favicon, the `favicon.ico` image can be placed in the assets directory.

Overview of the Page Layout

The callback in the `index.py` file takes as input the app URL and outputs different layouts depending upon the layout:

```
# Update page
# # # # # # # # #
@app.callback(Output('page-content', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):
    if pathname == '/cc-travel-report' or pathname == '/cc-travel-report/overview-birst
        return layout_birst_category
    elif pathname == '/cc-travel-report/overview-ga/':
        return layout_ga_category
    elif pathname == '/cc-travel-report/paid-search/':
        return layout_paid_search
    elif pathname == '/cc-travel-report/display/':
        return layout_display
    elif pathname == '/cc-travel-report/publishing/':
        return layout_publishing
    elif pathname == '/cc-travel-report/metasearch-and-travel-ads/':
        return layout_metasearch
    else:
        return noPage
```

dashboard_callback_display_page.py hosted with ❤ by GitHub view raw

Code Block 7: Callback in index.py File

The `layouts.py` file contains the following:

- Normal python import statements
- Statement to read in the data CSV file into pandas data frame
- Definition of the data table columns needed in the different variations of the layouts

- Definition of each page layout section
- Definition of the `noPage` layout

The beginning of the `layouts.py` file is as follows:

```
import dash_core_components as dcc
import dash_html_components as html
import dash_table
from components import Header, print_button
from datetime import datetime as dt
from datetime import date, timedelta
import pandas as pd


# Read in Travel Report Data
df = pd.read_csv('data/performance_analytics_cost_and_ga_metrics.csv')

df.rename(columns={
 'Travel Product': 'Placement type',
  'Spend - This Year': 'Spend TY',
  'Spend - Last Year': 'Spend LY',
  'Sessions - This Year': 'Sessions - TY',
  'Sessions - Last Year': 'Sessions - LY',
  'Bookings - This Year': 'Bookings - TY',
  'Bookings - Last Year': 'Bookings - LY',
  'Revenue - This Year': 'Revenue - TY',
  'Revenue - Last Year': 'Revenue - LY',
  }, inplace=True)


df['Date'] = pd.to_datetime(df['Date'])
current_year = df['Year'].max()

dt_columns = ['Placement type', 'Spend TY', 'Spend - LP', 'Spend PoP (Abs)', 'Spend PoP
                        'Sessions - TY', 'Sessions - LP', 'Sessions - LY', 'Sessions Po
                        'Bookings - TY', 'Bookings - LP', 'Bookings PoP (%)', 'Bookings
                        'Revenue - TY', 'Revenue - LP', 'Revenue PoP (Abs)', 'Revenue P

conditional_columns = ['Spend_PoP_abs_conditional', 'Spend_PoP_percent_conditional', 'S
'Sessions_PoP_percent_conditional', 'Sessions_YoY_percent_conditional',
'Bookings_PoP_abs_conditional', 'Bookings_YoY_abs_conditional', 'Bookings_PoP_percent_c
'Revenue_PoP_abs_conditional', 'Revenue_YoY_abs_conditional', 'Revenue_PoP_percent_conc

dt_columns_total = ['Placement type', 'Spend TY', 'Spend - LP', 'Spend PoP (Abs)', 'Spe
                        'Sessions - TY', 'Sessions - LP', 'Sessions - LY', 'Sessions Po
                        'Bookings - TY', 'Bookings - LP', 'Bookings PoP (%)', 'Bookings
```

```
                        'Revenue - TY', 'Revenue - LP', 'Revenue PoP (Abs)', 'Revenue P
                        'Spend_PoP_abs_conditional', 'Spend_PoP_percent_conditional',
'Sessions_PoP_percent_conditional', 'Sessions_YoY_percent_conditional',
'Bookings_PoP_abs_conditional', 'Bookings_YoY_abs_conditional', 'Bookings_PoP_percent_c
'Revenue_PoP_abs_conditional', 'Revenue_YoY_abs_conditional', 'Revenue_PoP_percent_cond

df_columns_calculated = ['Placement type', 'CPS - TY',
                        'CPS - LP', 'CPS PoP (Abs)', 'CPS PoP (%)',
                        'CPS - LY',  'CPS YoY (Abs)',  'CPS YoY (%)',
                        'CVR - TY',
                        'CVR - LP', 'CVR PoP (Abs)', 'CVR PoP (%)',
                        'CVR - LY',  'CVR YoY (Abs)',  'CVR YoY (%)',
                        'CPA - TY',
                        'CPA - LP', 'CPA PoP (Abs)', 'CPA PoP (%)',
                        'CPA - LY', 'CPA YoY (Abs)',  'CPA YoY (%)']

conditional_columns_calculated_calculated = ['CPS_PoP_abs_conditional', 'CPS_PoP_percen
'CVR_PoP_abs_conditional', 'CVR_PoP_percent_conditional', 'CVR_YoY_abs_conditional', 'C
'CPA_PoP_abs_conditional', 'CPA_PoP_percent_conditional', 'CPA_YoY_abs_conditional', 'C
```

Code Block 8: Beginning of layouts.py File

Each page layout section contains the following elements:

- Call to `Header()` which is imported, so one can have common elements (logo, brand name, etc.) across multiple apps.
- Data Range element
- Header Bar
- Radio Button to select condensed vs. complete data frame
- First data table
- Download button
- Second data table
- Graphs

A simplified code block for one layout section in the `layouts.py` is below:

```
######################## START Paid Search Layout ########################
layout_paid_search =  html.Div([
```

```
    html.Div([
        # CC Header
        Header(),
        # Date Picker
        html.Div([
            dcc.DatePickerRange(
              id='my-date-picker-range-paid-search',
              min_date_allowed=dt(2018, 1, 1),
              max_date_allowed=df['Date'].max().to_pydatetime(),
              initial_visible_month=dt(current_year,df['Date'].max().to_pydatetime().mo
              start_date=(df['Date'].max() - timedelta(6)).to_pydatetime(),
              end_date=df['Date'].max().to_pydatetime(),
            ),
            html.Div(id='output-container-date-picker-range-paid-search')
            ], className="row ", style={'marginTop': 30, 'marginBottom': 15}),
        # Header Bar
        html.Div([
          html.H6(["Paid Search"], className="gs-header gs-text-header padded",style={
          ]),
        # Radio Button
        html.Div([
          dcc.RadioItems(
            options=[
                {'label': 'Condensed Data Table', 'value': 'Condensed'},
                {'label': 'Complete Data Table', 'value': 'Complete'},
            ], value='Condensed',
            labelStyle={'display': 'inline-block', 'width': '20%', 'margin':'auto', 'ma
            id='radio-button-paid-search'
            )]),
        # First Data Table
        html.Div([
            dash_table.DataTable(
                id='datatable-paid-search',
                columns=[{"name": i, "id": i, 'deletable': True} for i in dt_columns]
                + [{"name": j, "id": j, 'hidden': 'True'} for j in conditional_columns]
                editable=True,
                n_fixed_columns=2,
                style_table={'maxWidth': '1500px'},
                row_selectable="multi",
                selected_rows=[0],
                ),
            ], className=" twelve columns"),
        # Download Button
        html.Div([
          html.A(html.Button('Download Data', id='download-button'), id='download-link
          ]),
        # Second Data Table
```

```
        html.Div([
            dash_table.DataTable(
              id='datatable-paid-search-2',
              columns=[{"name": i, "id": i} for i in df_columns_calculated] +
              [{"name": k, "id": k, 'hidden': 'True'} for k in conditional_columns_calc
              editable=True,
              n_fixed_columns=1,
              css=[{'selector': '.dash-cell div.dash-cell-value', 'rule': 'display: inl
              style_table={'maxWidth': '1500px'},
                ),
            ], className=" twelve columns"),
        # GRAPHS
        html.Div([
            html.Div([
              dcc.Graph(id='paid-search'),
              ], className=" twelve columns"
              )
            ], className="row ")
        ], className="subpage")
    ], className="page")
```

The above code does not include conditional formatting, which I will go into below.

Local Testing of the App

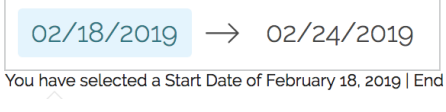
Following the user guide, after one has created the `index.py` and `app.py` files, one invokes the app with

```
$ python app.py
...Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

and visit http:127.0.0.1:8050/cc-travel-report/paid-search/ in your web browser.

. . .

7. Building the Date Selector Element



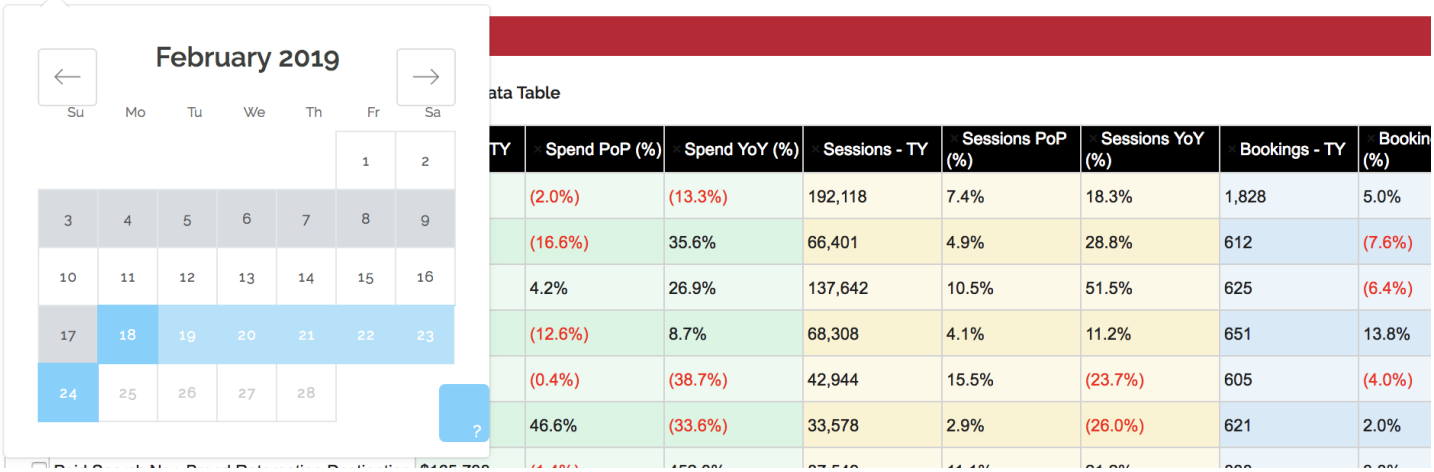


Figure 4: Date Selector Element

The date selector element provided a means to update the data presented in both data tables, as well as the downloaded data link. I also wanted the date selector to provide feedback on the dates selected by the user.

I simply chose January 1, 2018 as the minimum date allowed (`min_date_allowed`); The maximum date allowed was based upon the maximum date in the CSV data file (`max_date_allowed`); the initial month presented in the date selected was based upon the maximum date in the CSV data file (`initial_visible_month`); the initial start date was the maximum date minus 6 days (`start_date`); and the initial end date was the maximum date.

The following code is located in the `layouts.py` file and needs to be repeated for each "page" of the Dashboard app.

```
# Date Picker
html.Div([
    dcc.DatePickerRange(
      id='my-date-picker-range-paid-search',
      min_date_allowed=dt(2018, 1, 1),
      max_date_allowed=df['Date'].max().to_pydatetime(),
      initial_visible_month=dt(current_year,df['Date'].max().to_pydatetime().month, 1),
      start_date=(df['Date'].max() - timedelta(6)).to_pydatetime(),
      end_date=df['Date'].max().to_pydatetime(),
    ),
    html.Div(id='output-container-date-picker-range-paid-search')
    ], className="row ", style={'marginTop': 30, 'marginBottom': 15}),
```

dashboard_datepickerrange.py hosted with ❤ by **GitHub** **view raw**

Code Block 10: Html Element for Date Picker in layouts.py File

I used a callback and a function, `update_output` to provide feedback to the user. Specifically, I wanted to provide text feedback on the dates selected, the number of days selected, as well as the dates in the corresponding previous period. For instance, if a user had selected week numbers 5 and 6 of 2019 using the date range selector, then the corresponding past period would be week numbers 3 and 4 of 2019. The `update_output` function below provides this functionality.

```
#### Date Picker Callback
@app.callback(Output('output-container-date-picker-range-paid-search', 'children'),
        [Input('my-date-picker-range-paid-search', 'start_date'),
         Input('my-date-picker-range-paid-search', 'end_date')])
def update_output(start_date, end_date):
        string_prefix = 'You have selected '
        if start_date is not None:
                start_date = dt.strptime(start_date, '%Y-%m-%d')
                start_date_string = start_date.strftime('%B %d, %Y')
                string_prefix = string_prefix + 'a Start Date of ' + start_date_string
        if end_date is not None:
                end_date = dt.strptime(end_date, '%Y-%m-%d')
                end_date_string = end_date.strftime('%B %d, %Y')
                days_selected = (end_date - start_date).days
                prior_start_date = start_date - timedelta(days_selected + 1)
                prior_start_date_string = datetime.strftime(prior_start_date, '%B %d, %
                prior_end_date = end_date - timedelta(days_selected + 1)
                prior_end_date_string = datetime.strftime(prior_end_date, '%B %d, %Y')
                string_prefix = string_prefix + 'End Date of ' + end_date_string + ', t
                prior_start_date_string + ' | End Date: ' + prior_end_date_string + '.
        if len(string_prefix) == len('You have selected: '):
                return 'Select a date to see it displayed here'
        else:
                return string_prefix
```

dashboard_datepicker_callback.py hosted with ❤ by **GitHub** **view raw**

Code Block 11: Callback and Function for Date Picker in layouts.py file

The date picker element provides input to the callbacks for the first data table, the second data table, the download link, as well as the set of graphs below the data tables. I will go into detail how each of these callbacks work, with their associated outputs.

Adding a correction to CSS so that the Date Picker is not hidden behind the data tables

One issue I had to correct was that the data tables obscured the date picker element, so I had to correct the CSS accordingly.

```
#datatable-paid-search > div > div > div.row.row-1 > div.cell.cell-1-0.dash-fixed-columr
    flex: 0 0 auto;
    left: 0;
    position: sticky;
    z-index: 0;
}
```

dashboard_datepicker.css hosted with ❤ by GitHub view raw

Code Block 12: CSS for change z-index of data table in custom.css file

. . .

8. Building the First Data Table

Figure 5: First Data Table with a Condensed View

The first data table in the dashboard presents metrics such as spend (the cost associated with a given advertising product), website sessions, bookings (transactions), and revenue. These metrics are aggregated depending upon the dates selected in the date selected and typically present data for the current year for the date range selected, data for the previous corresponding period, data for last year, as well as percent and absolute differences between these periods.

Changing the Dates Presented in the Data Table based upon the Date Selection

The main functionality I wanted for the first data table is making it interactive, whereby the data presented changes according in the date selected. To start with, the data table element needs to be included in the `layouts.py` file as in the (simplified) code below:

```
# First Data Table
html.Div([
    dash_table.DataTable(
        id='datatable-paid-search',
        columns=[{"name": i, "id": i, 'deletable': True} for i in dt_columns],
        editable=True,
        n_fixed_columns=2,
        style_table={'maxWidth': '1500px'},
        row_selectable="multi",
        selected_rows=[0],
        style_cell = {"fontFamily": "Arial", "size": 10, 'textAlign': 'left'}
        )
    ], className=" twelve columns")
```

dashboard_simplified_first_data_table.py hosted with ❤ by **GitHub** **view raw**

Code Block 13: Data Table Component in layouts.py File

The different parameters for the data table include:

- `id` : identifier so that the data table can be referenced by the callbacks.
- `columns` : the columns presented in the table; `deletable` gives the ability of the user to delete the column while using the app.
- `editable=True` gives the user the ability to change the data in the table while using the app.
- `n_fixed_columns=2` freezes the first two columns (the checkboxes and the placement type in our case), so when a user scrolls across the complete data table, the user can still view the check boxes and the placement type.
- `style_table` allows CSS styles to be applied to the table.
- `row_selectable='multi'` allows the user to select multiple rows via the checkboxes. This will enable updating of the graphs below based upon the selection.
- `selected_rows=0` contains the index of the rows that are selected initially (and presented in the graphs below).

- `style_cell` allows CSS styles to be applied to the table cells.

I will add additional parameters in a subsequent step so we can have conditional style formatting. Number formatting such as dollar signs, commas, and percent signs are added in pandas and defined as a series of formatters.

I should note that I do not specify the `data` parameter for the data table in the `layouts.py` file. Rather, the `data` parameter for the data table will come from the callback:

```
# Callback and update first data table
@app.callback(Output('datatable-paid-search', 'data'),
        [Input('my-date-picker-range-paid-search', 'start_date'),
         Input('my-date-picker-range-paid-search', 'end_date')])
def update_data_1(start_date, end_date):
        data_1 = update_first_datatable(start_date, end_date, 'Paid Search', 'Placement
        return data_1
```

dashboard_first_datatable_callback.py hosted with ❤ by **GitHub** **view raw**

Code Block 14: Callback for First Data Table in layouts.py File

Calculating Changes in the Metrics, as well as the calculating cost-per-session, conversion rate, and cost-per-acquisition.

Since we need to calculate changes in the metrics, as well as CPA, CPS, and conversion rate depending upon the dates selected "on-the-fly," I push these calculations into a separate file, `functions.py`. I can then re-use these functions for the different Dashboard pages. I also define formatting functions in this file:

```
# Define Formatters
def formatter_currency(x):
        return "${:,.0f}".format(x) if x >= 0 else "(${:,.0f})".format(abs(x))

def formatter_currency_with_cents(x):
        return "${:,.2f}".format(x) if x >= 0 else "(${:,.2f})".format(abs(x))

def formatter_percent(x):
        return "{:,.1f}%".format(x) if x >= 0 else "({:,.1f}%)".format(abs(x))

def formatter_percent_2_digits(x):
        return "{:,.2f}%".format(x) if x >= 0 else "({:,.2f}%)".format(abs(x))

def formatter_number(x):
```

```
        return "{:,.0f}".format(x) if x >= 0 else "({:,.0f})".format(abs(x))
```

dashboard_formatting_functions.py hosted with ❤ by GitHub view raw

Code Block 15: Data Formatters in functions.py file

The functions for both the first data table and the second data table are similar so I only present one here:

```
# First Data Table Update Function
def update_first_datatable(start_date, end_date, category, aggregation):
        if start_date is not None:
                start_date = dt.strptime(start_date, '%Y-%m-%d')
                start_date_string = start_date.strftime('%Y-%m-%d')
        if end_date is not None:
                end_date = dt.strptime(end_date, '%Y-%m-%d')
                end_date_string = end_date.strftime('%Y-%m-%d')
        days_selected = (end_date - start_date).days

        prior_start_date = start_date - timedelta(days_selected + 1)
        prior_start_date_string = datetime.strftime(prior_start_date, '%Y-%m-%d')
        prior_end_date = end_date - timedelta(days_selected + 1)
        prior_end_date_string = datetime.strftime(prior_end_date, '%Y-%m-%d')

        if aggregation == 'Placement type':
                df1 = df[(df['Category'] == category)].groupby(['Date', aggregation]).
                df_by_date = df1[(df1['Date'] >= start_date_string) & (df1['Date'] <=
                df_by_date_prior = df1[(df1['Date'] >= prior_start_date_string) & (df1
                df_by_date_prior.rename(columns={'Spend TY' : 'Spend - LP', 'Sessions
                df_by_date_combined =  pd.merge(df_by_date, df_by_date_prior, on=[aggr
        elif aggregation == 'GA Category':
                df1 = df.groupby(['Date', aggregation]).sum()[columns].reset_index()
                df_by_date = df1[(df1['Date'] >= start_date_string) & (df1['Date'] <=
                df_by_date_prior = df1[(df1['Date'] >= prior_start_date_string) & (df1
                df_by_date_prior.rename(columns={'Spend TY' : 'Spend - LP', 'Sessions
                df_by_date_combined =  pd.merge(df_by_date, df_by_date_prior, on=[aggr
                df_by_date_combined.rename(columns={'GA Category':'Placement type'}, i
        elif aggregation == 'Birst Category':
                df1 = df.groupby(['Date', aggregation]).sum()[columns].reset_index()
                df_by_date = df1[(df1['Date'] >= start_date_string) & (df1['Date'] <=
                df_by_date_prior = df1[(df1['Date'] >= prior_start_date_string) & (df1
                df_by_date_prior.rename(columns={'Spend TY' : 'Spend - LP', 'Sessions
                df_by_date_combined =  pd.merge(df_by_date, df_by_date_prior, on=[aggr
                df_by_date_combined.rename(columns={'Birst Category':'Placement type']

        # Calculate Differences on-the-fly
        df_by_date_combined['Spend PoP (%)'] = np.nan
        df_by_date_combined['Spend YoY (%)'] = np.nan
```

```
df_by_date_combined['Spend YoY (%)'] = np.nan
df_by_date_combined['Sessions PoP (%)'] = np.nan
df_by_date_combined['Sessions YoY (%)'] = np.nan
df_by_date_combined['Bookings PoP (%)'] = np.nan
df_by_date_combined['Bookings YoY (%)'] = np.nan
df_by_date_combined['Revenue PoP (%)'] = np.nan
df_by_date_combined['Revenue YoY (%)'] = np.nan

df_by_date_combined['Spend_PoP_abs_conditional'] = df_by_date_combined['Spend
# Formatter
df_by_date_combined['Spend PoP (Abs)'] = df_by_date_combined['Spend PoP (Abs)

df_by_date_combined['Spend_PoP_percent_conditional'] = df_by_date_combined['Sp
        (((df_by_date_combined['Spend TY'] - df_by_date_combined['Spend - LP']
# Formatter
df_by_date_combined['Spend PoP (%)'] = np.where((df_by_date_combined['Spend TY
        df_by_date_combined['Spend PoP (%)'].apply(formatter_percent), df_by_d

df_by_date_combined['Spend_YoY_percent_conditional'] = df_by_date_combined['Sp
        ((df_by_date_combined['Spend TY'] - df_by_date_combined['Spend LY'])/d
# Formatter
df_by_date_combined['Spend YoY (%)'] = np.where((df_by_date_combined['Spend TY
        df_by_date_combined['Spend YoY (%)'].apply(formatter_percent), df_by_d



df_by_date_combined['Sessions_PoP_percent_conditional'] = df_by_date_combined[
        ((df_by_date_combined['Sessions - TY'] - df_by_date_combined['Sessions
# Formatter
df_by_date_combined['Sessions PoP (%)'] = np.where((df_by_date_combined['Sessi
        df_by_date_combined['Sessions PoP (%)'].apply(formatter_percent), df_b

df_by_date_combined['Sessions_YoY_percent_conditional'] = df_by_date_combined[
        ((df_by_date_combined['Sessions - TY'] - df_by_date_combined['Sessions
# Formatter
df_by_date_combined['Sessions YoY (%)'] = np.where((df_by_date_combined['Sessi
        df_by_date_combined['Sessions YoY (%)'].apply(formatter_percent), df_b



df_by_date_combined['Bookings_PoP_abs_conditional'] = df_by_date_combined['Boo
# Formatter
df_by_date_combined['Bookings PoP (Abs)'] = df_by_date_combined['Bookings PoP

df_by_date_combined['Bookings_YoY_abs_conditional'] = df_by_date_combined['Boo
# Formatter
df_by_date_combined['Bookings YoY (Abs)'] = df_by_date_combined['Bookings YoY

df_by_date_combined['Bookings_PoP_percent_conditional'] = df_by_date_combined[
        (df_by_date_combined['Bookings - TY'] - df_by_date_combined['Bookings
```

```
        # Formatter
        df_by_date_combined['Bookings PoP (%)'] = np.where((df_by_date_combined['Booki
                df_by_date_combined['Bookings PoP (%)'].apply(formatter_percent), df_b

        df_by_date_combined['Bookings_YoY_percent_conditional'] = df_by_date_combined|
                (df_by_date_combined['Bookings - TY'] - df_by_date_combined['Bookings
        # Formatter
        df_by_date_combined['Bookings YoY (%)'] = np.where((df_by_date_combined['Booki
                df_by_date_combined['Bookings YoY (%)'].apply(formatter_percent), df_b


        df_by_date_combined['Revenue_PoP_abs_conditional'] = df_by_date_combined['Reve
        # Formatter
        df_by_date_combined['Revenue PoP (Abs)'] = df_by_date_combined['Revenue PoP (A

        df_by_date_combined['Revenue_YoY_abs_conditional'] = df_by_date_combined['Reve
        # Formatter
        df_by_date_combined['Revenue YoY (Abs)'] = df_by_date_combined['Revenue YoY (A

        df_by_date_combined['Revenue_PoP_percent_conditional'] = df_by_date_combined['
                (df_by_date_combined['Revenue - TY'] - df_by_date_combined['Revenue -
        # Formatter
        df_by_date_combined['Revenue PoP (%)'] = np.where((df_by_date_combined['Revenu
                df_by_date_combined['Revenue PoP (%)'].apply(formatter_percent), df_by

        df_by_date_combined['Revenue_YoY_percent_conditional'] = df_by_date_combined['
                (df_by_date_combined['Revenue - TY'] - df_by_date_combined['Revenue -
        # Formatter
        df_by_date_combined['Revenue YoY (%)'] = np.where((df_by_date_combined['Revenu
                df_by_date_combined['Revenue YoY (%)'].apply(formatter_percent), df_by

        # Format Numbers
        df_by_date_combined['Spend TY'] = df_by_date_combined['Spend TY'].apply(format
        df_by_date_combined['Spend - LP'] = df_by_date_combined['Spend - LP'].apply(fo
        df_by_date_combined['Spend LY'] = df_by_date_combined['Spend LY'].apply(format

        df_by_date_combined['Sessions - TY'] = df_by_date_combined['Sessions - TY'].ap
        df_by_date_combined['Sessions - LP'] = df_by_date_combined['Sessions - LP'].ap
        df_by_date_combined['Sessions - LY'] = df_by_date_combined['Sessions - LY'].ap
        df_by_date_combined['Bookings - TY'] = df_by_date_combined['Bookings - TY'].ap
        df_by_date_combined['Bookings - LP'] = df_by_date_combined['Bookings - LP'].ap
        df_by_date_combined['Bookings - LY'] = df_by_date_combined['Bookings - LY'].ap

        df_by_date_combined['Revenue - TY'] = df_by_date_combined['Revenue - TY'].appl
        df_by_date_combined['Revenue - LP'] = df_by_date_combined['Revenue - LP'].appl
        df_by_date_combined['Revenue - LY'] = df_by_date_combined['Revenue - LY'].appl
        # Rearrange the columns
        df_by_date_combined_df = df_by_date_combined[[
```

```
                'Placement type',
                'Spend TY', 'Spend - LP', 'Spend PoP (Abs)', 'Spend PoP (%)', 'Spend
                'Sessions - TY', 'Sessions - LP', 'Sessions PoP (%)', 'Sessions - LY
                'Bookings - TY', 'Bookings - LP', 'Bookings PoP (%)', 'Bookings PoP (
                'Revenue - TY', 'Revenue - LP', 'Revenue PoP (Abs)', 'Revenue PoP (%)
                # 'Spend_PoP_percent_conditional',
                ]]

    data_df = df_by_date_combined.to_dict("rows")
    return data_df
```

The flow between the data table element, the callback and the function can be portrayed as:

Figure 6: Flow for the first data table

A method to select either a condensed data table or the complete data table.

One of the features that I wanted for the data table was the ability to show a “condensed” version of the table as well as the complete data table. Therefore, I included a radio button in the `layouts.py` file to select which version of the table to present:

```
# Radio Button
html.Div([
  dcc.RadioItems(
    options=[
        {'label': 'Condensed Data Table', 'value': 'Condensed'},
        {'label': 'Complete Data Table', 'value': 'Complete'},
    ], value='Condensed',
    labelStyle={'display': 'inline-block', 'width': '20%', 'margin':'auto', 'marginTop
    id='radio-button-paid-search'
    )]),
```

dashboard_radio_button_layout.py hosted with ❤ by GitHub view raw

Code Block 17: Radio Button in layouts.py

The callback for this functionality takes input from the radio button and outputs the columns to render in the data table:

```
# Callback and update data table columns
@app.callback(Output('datatable-paid-search', 'columns'),
    [Input('radio-button-paid-search', 'value')])
def update_columns(value):
    if value == 'Complete':
        column_set=[{"name": i, "id": i, 'deletable': True} for i in columns_complete] -
    elif value == 'Condensed':
        column_set=[{"name": i, "id": i, "deletable": True} for i in columns_condensed]
    return column_set
```

dashboard_radiobutton_callback.py hosted with ❤ by GitHub view raw

Code Block 18: Callback for Radio Button in layouts.py File

This callback is a little bit more complicated since I am adding columns for conditional formatting (which I will go into below). Essentially, just as the callback below is changing the data presented in the data table based upon the dates selected using the callback statement, `Output('datatable-paid-search', 'data'` , this callback is changing the columns presented in the data table based upon the radio button selection using the callback statement, `Output('datatable-paid-search', 'columns'.`

Conditionally Color-Code Different Data Table cells

One of the features which the stakeholders wanted for the data table was the ability to have certain numbers or cells in the data table to be highlighted based upon a metric's

value; red for negative numbers for instance. However, conditional formatting of data table cells has three main issues.

- There is lack of formatting functionality in Dash Data Tables at this time.
- If a number is formatted prior to inclusion in a Dash Data Table (in pandas for instance), then data table functionality such as sorting and filtering does not work properly.
- There is a bug in the Dash data table code in which conditional formatting does not work properly.

I ended up formatting the numbers in the data table in pandas despite the above limitations. I discovered that conditional formatting in Dash does not work properly for formatted numbers (numbers with commas, dollar signs, percent signs, etc.). Indeed, I found out that there is a bug with the method described in the Conditional Formatting — Highlighting Cells section of the Dash Data Table User Guide:

```
dash_table.DataTable(
    data=df.to_dict('rows'),
    columns=[
        {'name': i, 'id': i} for i in df.columns
    ],
    style_data_conditional=[
        {
            'if': {
                'column_id': 'Region',
                'filter': 'Region eq "Montreal"'
            },
            'backgroundColor': '#3D9970',
            'color': 'white',
        },
        {
            'if': {
                'column_id': 'Humidity',
                'filter': 'Humidity eq num(20)'
            },
            'backgroundColor': '#3D9970',
            'color': 'white',
        },
        {
            'if': {
                'column_id': 'Temperature',
                'filter': 'Temperature > num(3.9)'
```

```
            },
            'backgroundColor': '#3D9970',
            'color': 'white',
        },
    ]
)
```

dashboard_conditional_formatting_highlighting_cells.py hosted with ❤ by GitHub view raw

Code Block 19: Conditional Formatting — Highlighting Cells

The cell for New York City temperature shows up as green even though the value is less than 3.9.* I've tested this in other scenarios and it seems like the conditional formatting for numbers only uses the integer part of the condition ("3" but not "3.9"). The filter for Temperature used for conditional formatting somehow truncates the significant digits and only considers the integer part of a number. I posted to the Dash community forum about this bug, and it has since been fixed in a recent version of Dash.

**This has since been corrected in the Dash Documentation.*

Conditional Formatting of Cells using Doppelganger Columns

Due to the above limitations with conditional formatting of cells, I came up with an alternative method in which I add "doppelganger" columns to both the pandas data frame and Dash data table. These doppelganger columns had either the value of the original column, or the value of the original column multiplied by 100 (to overcome the bug when the decimal portion of a value is not considered by conditional filtering). Then, the doppelganger columns can be added to the data table but are hidden from view with the following statements:

```
dt_columns = ['Placement type', 'Spend TY', 'Spend - LP', 'Spend PoP (Abs)', 'Spend PoP
                        'Sessions - TY', 'Sessions - LP', 'Sessions - LY', 'Sessions Po
                        'Bookings - TY', 'Bookings - LP', 'Bookings PoP (%)', 'Bookings
                        'Revenue - TY', 'Revenue - LP', 'Revenue PoP (Abs)', 'Revenue P

conditional_columns = ['Spend_PoP_abs_conditional', 'Spend_PoP_percent_conditional', 'S
'Sessions_PoP_percent_conditional', 'Sessions_YoY_percent_conditional',
'Bookings_PoP_abs_conditional', 'Bookings_YoY_abs_conditional', 'Bookings_PoP_percent_c
'Revenue_PoP_abs_conditional', 'Revenue_YoY_abs_conditional', 'Revenue_PoP_percent_cond

# Below is placed in definition of data table
columns=[{"name": i, "id": i, 'deletable': True} for i in dt_columns]
                + [{"name": j, "id": j, 'hidden': 'True'} for j in conditional_columns]
```

dashboard_column_definition.py hosted with ❤ by **GitHub** **view raw**

Code Block 20: Adding Doppelganger Columns

Then, the conditional cell formatting can be implemented using the following syntax:

```
style_cell_conditional=[{'if': {'column_id': 'Revenue YoY (%)',
                         'filter': 'Revenue_YoY_percent_conditional < num(0)'},
                         'color': 'red'}]
```

dashboard_style_cell_conditional.py hosted with ❤ by **GitHub** **view raw**

Code Block 21: Conditional Cell Formatting

Essentially, the filter is applied on the "doppelganger" column, `Revenue_YoY_percent_conditional` (filtering cells in which the value is less than 0). However, the formatting is applied on the corresponding "real" column, `Revenue YoY (%)` . One can imagine other usages for this method of conditional formatting; for instance, highlighting outlier values.

The complete statement for the data table is below (with conditional formatting for odd and even rows, as well highlighting cells that are above a certain threshold using the doppelganger method):

```
html.Div([
    dash_table.DataTable(
        id='datatable-paid-search',
        columns=[{"name": i, "id": i, 'deletable': True} for i in dt_columns]
        + [{"name": j, "id": j, 'hidden': 'True'} for j in conditional_columns],
        editable=True,
        n_fixed_columns=2,
        style_table={'maxWidth': '1500px'},
        row_selectable="multi",
        selected_rows=[0],
        style_cell = {"fontFamily": "Arial", "size": 10, 'textAlign': 'left'},
        css=[{'selector': '.dash-cell div.dash-cell-value', 'rule': 'display: inline; w
        style_cell_conditional=[{'if': {'row_index': 'odd'}, 'backgroundColor': '#D5DBD
              + [{'if': {'column_id': c}, 'backgroundColor': '#EAFAF1'} for c in ['Sper
              + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#D5F5
              + [{'if': {'column_id': c}, 'backgroundColor': '#FEF9E7'} for c in ['Sess
              + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#FCF3
              + [{'if': {'column_id': c}, 'backgroundColor': '#EBF5FB'} for c in ['Book
              + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#D6EA
              + [{'if': {'column_id': c},'backgroundColor': '#F4ECF7'} for c in ['CVR -
              + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#E8DA
```

```
                + [{'if': {'column_id': c}, 'backgroundColor': '#FDEDEC' } for c in ['CPA
                + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#FADE
                + [{'if': {'column_id': c},'backgroundColor': '#F6DDCC'} for c in ['CPS -
                + [{'if': {'column_id': c, 'row_index': 'odd'}, 'backgroundColor': '#E598
                + [{'if': {'column_id': c}, 'minWidth': '0px', 'maxWidth': '80px', 'white
                'Sessions YoY (%)', 'Bookings - TY', 'Bookings - LP', 'Bookings PoP (%)',
                + [{'if': {'column_id': 'Spend PoP (Abs)', 'filter': 'Spend_PoP_abs_condi
                + [{'if': {'column_id': 'Spend PoP (%)', 'filter': 'Spend_PoP_percent_cor
                + [{'if': {'column_id': 'Spend YoY (%)', 'filter': 'Spend_YoY_percent_cor
                + [{'if': {'column_id': 'Sessions PoP (%)', 'filter': 'Sessions_PoP_perce
                + [{'if': {'column_id': 'Sessions YoY (%)', 'filter': 'Sessions_YoY_perce
                + [{'if': {'column_id': 'Bookings PoP (Abs)', 'filter': 'Bookings_PoP_abs
                + [{'if': {'column_id': 'Bookings YoY (Abs)', 'filter': 'Bookings_YoY_abs
                + [{'if': {'column_id': 'Bookings PoP (%)', 'filter': 'Bookings_PoP_perce
                + [{'if': {'column_id': 'Bookings YoY (%)', 'filter': 'Bookings_YoY_perce
                + [{'if': {'column_id': 'Revenue PoP (Abs)', 'filter': 'Revenue_PoP_abs_c
                + [{'if': {'column_id': 'Revenue YoY (Abs)', 'filter': 'Revenue_YoY_abs_c
                + [{'if': {'column_id': 'Revenue PoP (%)', 'filter': 'Revenue_PoP_percent
                + [{'if': {'column_id': 'Revenue YoY (%)', 'filter': 'Revenue_YoY_percent
                style_header={'backgroundColor': 'black','color': 'white'},
        )
```

dashboard_datatable_with_conditional_formating.py hosted with ❤ by GitHub view raw

Code Block 22: Data Table with Conditional Formatting

I describe the method to update the graphs using the selected rows in the data table below.

. . .

9. Building the Download Data link

Figure 7: Download data link

One of the features I wanted for the dashboard was the ability to download the data that was presented in the data tables. Specifically, I wanted the downloaded data to be updated according to the dates selected (and presented in the data tables). In addition, since the dates are not displayed, it was necessary to add the dates to the downloaded

file. Moreover, even though the data is formatted in the data tables (with `$`, `%`, and thousand comma separators, I wanted the downloaded data to be free of formatting.

There were two critical Plotly community threads which helped me to build this functionality at Allow users to dowload [sic] an Excel in a click and Allowing users to download CSV on click.

The download data functionality was implemented using the following:

- There is simply a placeholder for the download button in the `layouts.py` file.

```
# Download Button
html.Div([
  html.A(html.Button('Download Data', id='download-button'), id='download-link-paid-sear
  ]),
```

dashboard_download_link.py hosted with ❤ by **GitHub** **view raw**

Code Block 23: Placeholder for Download Button

- You will need the following modules for the download link to work properly:

```
import io
import xlsxwriter
import flask
from flask import send_file
```

dashboard_download_link_import.py hosted with ❤ by **GitHub** **view raw**

Code Block 24: Import Statements to Enable Download Functionality

- The callback for the excel download is placed in the `callbacks.py` file. The callback is taking the `start_date` and the `end_date` from the date picker element and outputting a file reference to the download link.
- The `update_link` function is updating the URL link to serve, based upon the start and end dates.
- The function `download_excel_1` takes the URL value and splits it up back into `start_date` and `end_date` so that I can name the `filename` based upon the start and end dates, as well as the current date.

```
# Callback for excel download
@app.callback(
```

```
@app.callback(
    Output('download-link-paid-search-1', 'href'),
    [Input('my-date-picker-range-paid-search', 'start_date'),
         Input('my-date-picker-range-paid-search', 'end_date')])
def update_link(start_date, end_date):
        return '/cc-travel-report/paid-search/urlToDownload?value={}/{}'.format(dt.str
@app.server.route("/cc-travel-report/paid-search/urlToDownload")
def download_excel_1():
    value = flask.request.args.get('value')
    #here is where I split the value
    value = value.split('/')
    start_date = value[0]
    end_date = value[1]

    filename = datestamp + '_paid_search_' + start_date + '_to_' + end_date + '.xlsx'
        # Dummy Dataframe
    # d = {'col1': [1, 2], 'col2': [3, 4]}
    # df = pd.DataFrame(data=d)

    buf = io.BytesIO()
    excel_writer = pd.ExcelWriter(buf, engine="xlsxwriter")
    download_1 = update_first_download(start_date, end_date, 'Paid Search', 'Placement
    download_1.to_excel(excel_writer, sheet_name="sheet1", index=False)
    # df.to_excel(excel_writer, sheet_name="sheet1", index=False)
    excel_writer.save()
    excel_data = buf.getvalue()
    buf.seek(0)

    return send_file(
        buf,
        mimetype = 'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet',
        attachment_filename=filename,
        as_attachment=True,
        cache_timeout=0
    )
```

dashboard download link callback.py hosted with ❤ by GitHub view raw

Code Block 25: Callback and function for excel download data link

- The function `download_excel_1` also calls another function, `update_first_download`, which is very similar to the `update_first_download` function presented above, except that the data is not formatted.
- The downloaded Excel file is named according to the product name and the start and end dates selected, as well as the current date. Columns for the start and end dates and other appropriate dates are added to the downloaded data file.

Figure 8: Snap shot of the downloaded data.

. . .

10. Building the Second Data Table

Figure 9: Data Table with Calculated Metrics

The second data table featured metrics such as cost-per-session, conversion rate and and cost-per-session which are calculated “on-the-fly,” depending upon the dates selected in the date range selected.

The second data table is created is a similar manner to the first data table so the details are left out of this tutorial, but the complete files are on Github.

. . .

11. Updating Graphs by Selecting Rows in a Dash Data table

Figure 10: Digital Marketing Metric Graphs

The graphs below the data tables display metrics aggregated by week, and include the current year's data, last year's data, as well as the percentage changes between the two.

Figure 11: Diagram of callback and function to update graphs

The steps to update the graphs are:

1. Set placeholder in the `layouts.py` file.
2. Create callback for each set of graphs, for each page of the Dashboard in the `callbacks.py` file.
3. Build a function which filters the complete list of products by the list of products selected in the Dash data table, and subsequently creates a pandas data frame based upon this selection.
4. The filtered data frame is then passed to another function, `update_graph` that (a) calculates metrics such as cost-per-session, conversion rate and cost-per session, and (b) produces the output for the graphs.

These steps are detailed below. *Step 4 is detailed in Section 12, "Updating the Graphs and Calculating metrics on the fly."*

Step 1

In the `layouts.py` file, there is simply a placeholder for the graphs:

```
1  html.Div([
2      dcc.Graph(id='paid-search'),
3      ], className=" twelve columns"
4      )
```

dashboard_graph_placeholder.py hosted with ❤ by **GitHub** **view raw**

Code Block 26: Placeholder for Graphs

Step 2

There is a callback for each set of graphs on each page in the `callbacks.py` file:

```
1  # Callback for the Graphs
2  @app.callback(
3     Output('paid-search', 'figure'),
4     [Input('datatable-paid-search', "selected_rows"),
5     Input('my-date-picker-range-paid-search', 'end_date')])
```

dashboard_graph_callback.py hosted with ❤ by **GitHub** **view raw**

Code Block 27: Callback for Graphs

The callback takes input from the first data table, as well as input from the date picker, and outputs to the `dcc.Graph` element with the `id`, `paid-search`.

Step 3

One of the challenges I had had was to figure out a way of updating the graphs depending upon the products selected in the data table.

In the code for the first data table, it was necessary to set the parameters for `row_selectable` to `multi` and `selected_rows=[0]`. The former parameter enables checkboxes next to each row in the data table, whereas the latter parameter ensures that the selected rows has an initial value. The parameter, `n_fixed_columns` freezes the first two rows in the data table so that they are visible when a user enables the complete data frame, (thus exposing all of the available columns).

Here is a simplified code block for the first data table (which is in the `layouts.py` file):

```
dash_table.DataTable(
    id='datatable-paid-search',
    columns=[{"name": i, "id": i, 'deletable': True} for i in dt_columns],
    n_fixed_columns=2,
    row_selectable="multi",
    selected_rows=[0],
    )
```

dashboard_first_datatable_simplified.py hosted with ❤ by **GitHub** **view raw**

Code Block 28: Simplified Data Table Definition

Hence, the callback for the graphs acquire the `selected_rows` . In the function to update the graphs, `update_paid_search` , a list of products is built by filtering the original data frame by the dashboard page category (Paid Search in this case) and getting a complete list of unique placement types. Then, a `for` loop filters this list by the list of products selected in the data table. Subsequently, a filtered data frame, `filtered_df` is created by filtering the original data frame by the list of selected products and performing a pandas `groupby` and summing the spend, sessions, bookings and revenue columns. The callback and function, `update_paid_search` , for the graphs is presented below:

```
# Callback for the Graphs
@app.callback(
   Output('paid-search', 'figure'),
   [Input('datatable-paid-search', "selected_rows"),
   Input('my-date-picker-range-paid-search', 'end_date')])
def update_paid_search(selected_rows, end_date):
        travel_product = []
        travel_product_list = df[(df['Category'] == 'Paid Search')]['Placement type'].u
        for i in selected_rows:
                travel_product.append(travel_product_list[i])
                # Filter by specific product
        filtered_df = df[(df['Placement type'].isin(travel_product))].groupby(['Year',
        fig = update_graph(filtered_df, end_date)
        return fig
```

dashboard_graph_callback_and_function.py hosted with ❤ by **GitHub** **view raw**

Code Block 29: Callback and Update Function for Graphs

• • •

12. Updating the Graphs and Calculating metrics on the fly

The graphs for each metric are aggregated by week and by the products that are selected in the data table (as detailed above).

The features I wanted for the graphs and for the update function included:

- Since the graphs can depict metrics on more than one product at a time, the update function needs to calculate the metrics on-the-fly.
- The graph for each metric should include the value for this year, the value for last year, as well as the year-to-year percentage difference.These should be overlaid on the same graph, with the values being line graphs, and the year-to-year change being bar graphs.
- I wanted the zoom function to work across all of the graphs simultaneously, such that zooming in on one graph, zooms in the other graphs at the same zoom level.

The complete `update_graph` function (which resides in the `functions.py` file is below:

```
def update_graph(filtered_df, end_date):
    if end_date is not None:
        end_date = dt.strptime(end_date, '%Y-%m-%d')
        end_date_string = end_date.strftime('%Y-%m-%d')
    if end_date_string <= '2018-12-29':
        current_year = 2018
    else:
        current_year = 2019

    # Calulate YoY Differences
    filtered_df['Spend YoY (%)'] = ((filtered_df['Spend TY'] - filtered_df['Spend LY']
    filtered_df['Sessions YoY (%)'] = ((filtered_df['Sessions - TY'] - filtered_df['Se
    filtered_df['Bookings - % - PY'] = ((filtered_df['Bookings - TY'] - filtered_df['B
    filtered_df['Revenue - % - PY'] = ((filtered_df['Revenue - TY'] - filtered_df['Rev

    # Calculate CPS, CR, CPA
    filtered_df['CPS - TY'] = np.nan
    filtered_df['CPS - LY'] = np.nan
    filtered_df['% YoY_CPS'] = np.nan

    filtered_df['CVR - TY'] = np.nan
    filtered_df['CVR - LY'] = np.nan
    filtered_df['CVR YoY (Abs)'] = np.nan

```

```
    filtered_df['CPA - TY'] = np.nan
    filtered_df['CPA - LY'] = np.nan
    filtered_df['% YoY_CPA'] = np.nan

    filtered_df['CPS - TY'] = np.where((filtered_df['Spend TY'] != 0) &  (filtered_df|
    filtered_df['CPS - LY'] = np.where((filtered_df['Spend LY'] != 0) &  (filtered_df|
    filtered_df['% YoY_CPS'] =  np.where((filtered_df['CPS - TY'] != 0) &  (filtered_c

    filtered_df['CVR - TY'] = np.where(((filtered_df['Bookings - TY'] != 0) & (filtere
    filtered_df['CVR - LY'] = np.where(((filtered_df['Bookings - LY'] != 0) & (filtere
    filtered_df['CVR YoY (Abs)'] =  np.where((filtered_df['CVR - TY'].notnull() & filt

    filtered_df['CPA - TY'] = np.where((filtered_df['Spend TY'] != 0) & (filtered_df['
    filtered_df['CPA - LY'] = np.where((filtered_df['Spend LY'] != 0) & (filtered_df['
    filtered_df['% YoY_CPA'] =  np.where((filtered_df['CPA - TY'] != 0) & (filtered_df


    # Sessions Graphs
    sessions_ty = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['Sessions - TY'],
      text='Sessions - TY'
    )
    sessions_ly = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year-1)]['Sessions - TY'],
      text='Sessions - LY'
    )
    sessions_yoy = go.Bar(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['Sessions YoY (%)'],
      text='Sessions YoY (%)', opacity=0.6
    )
    # Spend Graphs
    spend_ty = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['Spend TY'],
      text='Spend TY'
    )
    spend_ly = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year-1)]['Spend TY'],
      text='Spend LY'
    )
    spend_yoy = go.Bar(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['Spend YoY (%)'],
```

```
        text='Spend YoY (%)', opacity=0.6
    )
    # Bookings Graphs
    bookings_ty = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['Bookings - TY'],
        text='Bookings - TY'
    )
    bookings_ly = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year-1)]['Bookings - TY'],
        text='Bookings - LY'
    )
    bookings_yoy = go.Bar(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['Bookings - % - PY'],
        text='Bookings - % - PY', opacity=0.6
    )
    cpa_ty = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['CPA - TY'],
        text='CPA - TY'
    )
    cpa_ly = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year-1)]['CPA - TY'],
        text='CPA - LY'
    )
    cpa_yoy = go.Bar(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['% YoY_CPA'],
        text='% CPA - YoY', opacity=0.6
    )
    cps_ty = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['CPS - TY'],
        text='CPS - TY'
    )
    cps_ly = go.Scatter(
        x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year-1)]['CPS - TY'],
        text='CPS - LY'
    )
    cps_yoy = go.Bar(
        x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
        y=filtered_df[(filtered_df['Year'] == current_year)]['% YoY_CPS'],
        text='% CPS - YoY', opacity=0.6
    )
```

```
    )
    cr_ty = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['CVR - TY'],
      text='CVR - TY'
    )
    cr_ly = go.Scatter(
      x=filtered_df[(filtered_df['Year'] == current_year-1)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year-1)]['CVR - TY'],
      text='CVR - LY'
    )
    cr_yoy = go.Bar(
      x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
      y=filtered_df[(filtered_df['Year'] == current_year)]['CVR YoY (Abs)'],
      text='CVR YoY (Abs)', opacity=0.6
    )

    fig = tools.make_subplots(
      rows=6,
      cols=1,
      shared_xaxes=True,
      subplot_titles=(                                                    # Be s
        'Sessions',
        'Spend',
        'Bookings',
        'Cost per Acquisition',
        'CPS',
        'Conversion Rate'
        ))

    fig.append_trace(sessions_ty, 1, 1)         # 0
    fig.append_trace(sessions_ly, 1, 1)         # 1
    fig.append_trace(sessions_yoy, 1, 1)        # 2
    fig.append_trace(spend_ty, 2, 1)            # 3
    fig.append_trace(spend_ly, 2, 1)            # 4
    fig.append_trace(spend_yoy, 2, 1)           # 5
    fig.append_trace(bookings_ty, 3, 1)         # 6
    fig.append_trace(bookings_ly, 3, 1)         # 7
    fig.append_trace(bookings_yoy, 3, 1)        # 8
    fig.append_trace(cpa_ty, 4, 1)                      # 9
    fig.append_trace(cpa_ly, 4, 1)                      # 10
    fig.append_trace(cpa_yoy, 4, 1)                     # 11
    fig.append_trace(cps_ty, 5, 1)                      # 12
    fig.append_trace(cps_ly, 5, 1)                      # 13
    fig.append_trace(cps_yoy, 5, 1)                     # 14
    fig.append_trace(cr_ty, 6, 1)                       # 15
    fig.append_trace(cr_ly, 6, 1)                       # 16
    fig.append_trace(cr_yoy, 6, 1)                      # 17
```

```
167
168       # integer index below is the index of the trace
169       # yaxis indices below need to start from the number of total graphs + 1 since they
170       # overlaing and anchor axes correspond to the graph number
171
172       fig['data'][2].update(yaxis='y7')
173       fig['layout']['yaxis7'] = dict(overlaying='y1', anchor='x1', side='right', showgri
174
175       fig['data'][5].update(yaxis='y8')
176       fig['layout']['yaxis8'] = dict(overlaying='y2', anchor='x2', side='right', showgri
177
178       fig['data'][8].update(yaxis='y9')
179       fig['layout']['yaxis9'] = dict(overlaying='y3', anchor='x3', side='right', showgri
180
181       fig['data'][11].update(yaxis='y10')
182       fig['layout']['yaxis10'] = dict(overlaying='y4', anchor='x4', side='right', showgr
183
184       fig['data'][14].update(yaxis='y11')
185       fig['layout']['yaxis11'] = dict(overlaying='y5', anchor='x5', side='right', showgr
186
187       fig['data'][17].update(yaxis='y12')
188       fig['layout']['yaxis12'] = dict(overlaying='y6', anchor='x6', side='right', showgr
189
190       fig['layout']['xaxis'].update(title='Week of the Year' + ' - ' + str(current_year)
191       for i in fig['layout']['annotations']:
192         i['font'] = dict(size=12)
193       fig['layout'].update(
194         height= 1500,
195         # width=750,
196         showlegend=False,
197         xaxis=dict(
198           dtick=5,
199           ticklen=8,
200           tickwidth=2,
201           tickcolor='#000',
202           showgrid=True,
203           zeroline=True,
204           gridwidth=2
205       ),
206         )
207       updated_fig = fig
```

In order to "group" the graphs together, I utilized the subplot capability of Plotly detailed at https://plot.ly/python/subplots/. The `update_graph` function has the following main elements:

- On-the-fly calculation of the metrics.
- Each graph "trace" is defined using the `graph_objs` method in the Plotly library (remember to import this method via `import plotly.graph_objs as go`). Specifically, each graph is defined with the `go.Scatter` method:

```
sessions_ty = go.Scatter(
  x=filtered_df[(filtered_df['Year'] == current_year)]['Week'],
  y=filtered_df[(filtered_df['Year'] == current_year)]['Sessions - TY'],
  text='Sessions - TY'
)
```

dashboard_trace_assignment.py hosted with ❤ by **GitHub** **view raw**

Code Block 31: Trace Assignment for each Graph element

- The main graph figure and its parameters are defined when we call `tools.make_subplots`. During app development, I found that potential sources of error are not remembering to change the number of rows when you add additional traces (`rows=6`), as well as ensuring that you have the same number of titles as the number of graphs. Both of these might cause the app to fail.

```
fig = tools.make_subplots(
  rows=6,
  cols=1,
  shared_xaxes=True,
  subplot_titles=(    # Be sure to have same number of titles as number of graphs
    'Sessions',
    'Spend',
    'Bookings',
    'Cost per Acquisition',
    'CPS',
    'Conversion Rate'
    ))
```

dashboard_make_subplots.py hosted with ❤ by **GitHub** **view raw**

Code Block 32: make_subplots To Set Parameters for Subplots

- The different subplot "traces" are appended to the main graph figure with the statements such as `fig.append_trace(sessions_ty, 1, 1)`, where `sessions_ty` is the trace name defined above; the first number, `1` is the graph number and the last number is the column number (which is `1` in every case since we only have one

column). I have included commented-out numbers for each trace in order to assist me in the next step.

```
fig.append_trace(sessions_ty, 1, 1)             # 0
fig.append_trace(sessions_ly, 1, 1)             # 1
fig.append_trace(sessions_yoy, 1, 1)    # 2
fig.append_trace(spend_ty, 2, 1)                  # 3
fig.append_trace(spend_ly, 2, 1)                  # 4
fig.append_trace(spend_yoy, 2, 1)                 # 5
fig.append_trace(bookings_ty, 3, 1)             # 6
fig.append_trace(bookings_ly, 3, 1)             # 7
fig.append_trace(bookings_yoy, 3, 1)    # 8
fig.append_trace(cpa_ty, 4, 1)                    # 9
fig.append_trace(cpa_ly, 4, 1)                    # 10
fig.append_trace(cpa_yoy, 4, 1)                   # 11
fig.append_trace(cps_ty, 5, 1)                    # 12
fig.append_trace(cps_ly, 5, 1)                    # 13
fig.append_trace(cps_yoy, 5, 1)                   # 14
fig.append_trace(cr_ty, 6, 1)                       # 15
fig.append_trace(cr_ly, 6, 1)                       # 16
fig.append_trace(cr_yoy, 6, 1)                    # 17
```

dashboard_append_traces.py hosted with ❤ by **GitHub** **view raw**

Code Block 33: Appending Trace to the Figure

In order to overlay the year-to-year changes, I had to have several update statements, as well as adding new y-axes for each overlaid graph (the year-to-year change graphs). The syntax required for these overlaid graphs was a little tricky.

```
# integer index above is the index of the trace
# yaxis indices below need to start from the number of total graphs + 1 since they are
# overlaying and anchor axes correspond to the graph number

fig['data'][2].update(yaxis='y7')
fig['layout']['yaxis7'] = dict(overlaying='y1', anchor='x1', side='right', showgrid=Fal

fig['data'][5].update(yaxis='y8')
fig['layout']['yaxis8'] = dict(overlaying='y2', anchor='x2', side='right', showgrid=Fal

fig['data'][8].update(yaxis='y9')
fig['layout']['yaxis9'] = dict(overlaying='y3', anchor='x3', side='right', showgrid=Fal

fig['data'][11].update(yaxis='y10')
fig['layout']['yaxis10'] = dict(overlaying='y4', anchor='x4', side='right', showgrid=Fa

```

```
fig['data'][14].update(yaxis='y11')
fig['layout']['yaxis11'] = dict(overlaying='y5', anchor='x5', side='right', showgrid=Fa
fig['data'][17].update(yaxis='y12')
fig['layout']['yaxis12'] = dict(overlaying='y6', anchor='x6', side='right', showgrid=Fa
```

dashboard_overlaid_y_axes.py hosted with ❤ by **GitHub** **view raw**

Code Block 34: Code to Overlay Year-to-Year Change Graphs

- For instance, in the first update statement, the index number `2` in the statement `fig['data'][2]` refers to the third trace, `sessions_yoy` (since Python is zero-indexed).

The following figure hopefully assists in visualizing the indexing of the various elements of the combined graph.

Figure 12: Schematic of the Different Axis Indexes

- The y-axis name, `y7`, in the statement `yaxis='y7'` needs to be `7` since there are already 6 y-axes (one for each metric: sessions, spend, bookings, cpa, cps, and cr. For each metric, this year's and last year's data shares the same left y-axis). Hence, we need start numbering the right-side y axes at `7`.
- We need to assign the parameter for each of the additional y-axes using the `fig['layout']['yaxis']` statements. Specifically, the first right-side axis for the year-to-year changes in sessions overlaps with the other session traces in the first graph. Hence, we need to assign the parameters accordingly: `overlaying='y1'`, `anchor='x1'` and, of course, we need to assign it to the right side by setting `side='right'`. In addition, I update the title of the set of the graphs with the following:

```
fig['layout']['xaxis'].update(title='Week of the Year' + ' - ' + str(current_year))
```

dashboard_update_graph_title.py hosted with ❤ by **GitHub** **view raw**

Code Block 35: Update Title of Graphs

Finally, I need to update the overall figure layout with the following statement. I should note that there are several parameters below which are commented out since I am still experimenting with different parameters.

```
fig['layout'].update(
  height= 1500,
  # width=750,
  showlegend=False,
  xaxis=dict(
    # tickmode='linear',
    # ticks='outside',
    # tick0=1,
    dtick=5,
    ticklen=8,
    tickwidth=2,
    tickcolor='#000',
    showgrid=True,
    zeroline=True,
    # showline=True,
    # mirror='ticks',
    # gridcolor='#bdbdbd',
    gridwidth=2
)
```

dashboard_update_figure_layout.py hosted with ❤ by **GitHub** **view raw**

Code Block 36: Update Overall Figure Layout

. . .

13. Plotly Graph Tools

I should note that there are several helpful tools available in Dash to manipulate the graphs. If you hover over the top right-side of the graph figure, you can view these tools.

Figure 13: Plotly Graph Tools

With these tools you can:

- Drag to zoom in and double-click to return to the original graph.
- Drag the corners of a graph to zoom along one axis.
- Double-click to autoscale a single axis.
- Change the hover mode to compare data or investigate a single data point.

Certainly, one of the nicest tools is the ability to download a pic of the graph:

Figure 14: Downloaded image of the graphs

• • •

14. Deploying the Dashboard

I essentially followed the Deploying Dash Apps guide in the Dash User Guide, with a few changes. Specifically, I deployed the Dashboard on Heroku using the following steps:

- **Step 1**: I had already created a folder for the my dashboard code in a previous step.

```
$ mkdir dash_app_example
$ cd dash_app_example
```

dashboard_mkdir hosted with ❤ by **GitHub** **view raw**

Code Block 37: Mkdir and cd Code

- **Step 2**: Initialize the folder with `git` . Rather than use `venv` , I had previously set up a conda environment for Dash.

```
git init        # initializes an empty git repo
conda create --n dash
source activate dash
```

dashboard_git_init hosted with ❤ by **GitHub** **view raw**

Code Block 38: Init Git and Active Dash Conda Environment

And I had either installed the app's dependencies using either `conda` or `pip` :

```
pip install dash
pip install dash-renderer
pip install dash-core-components
pip install dash-html-components
pip install plotly
pip install pandas
pip install numpy
pip install xlsxwriter
pip install flask
pip install gunicorn
```

dashboard_pip_install hosted with ❤ by **GitHub** **view raw**

Code Block 39: Pip Installation of Dash and other Modules

You will also need a new dependency, `gunicorn` , for deploying the app (line 10 in the above code segment).

- **Step 3**: Initialize folder with an app (`app.py`), a `.gitignore` file, a `requirements.txt` file, and a `Procfile` file for deployment. We only need a minimal `app.py` since we are defining our app using multiple files. You need to create the following files in your project folder.

`app.py`

```
import dash
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
app = dash.Dash(__name__, external_stylesheets=external_stylesheets, url_base_pathname=
server = app.server
app.config.suppress_callback_exceptions = True
import dash_auth
# Keep this out of source code repository - save in a file or a database
VALID_USERNAME_PASSWORD_PAIRS = [
    ['[username]', '[password']
]
auth = dash_auth.BasicAuth(
    app,
    VALID_USERNAME_PASSWORD_PAIRS
)
```

dashboard_app.py hosted with ❤ by **GitHub** **view raw**

Code Block 40: app.py file (previously created)

`.gitignore`

```
*.pyc
.DS_Store
```

dashboard_gitignore hosted with ❤ by **GitHub** **view raw**

Code Block 41: .gitignore file

And a `Procfile` file:

```
web: gunicorn index:server
```

dashboard_Procfile hosted with ❤ by **GitHub** **view raw**

Code Block 42: Procfile file

You also need to generate a `requirements.txt` file. You can do this with:

```
pip freeze > requirements.txt
```

dashboard_pip_freeze hosted with ❤ by GitHub view raw

Code Block 43: Create new requirements.txt file

- **Step 4**: Initialize Heroku, add files to Git, and Deploy. Prior to initializing Heroku, you need to install the Heroku Command Line Interface (CLI) either with their installer available at https://devcenter.heroku.com/articles/heroku-cli or, on a Mac, using homebrew: `brew tap heroku/brew && brew install heroku`.

```
$ heroku create my-dash-app # change my-dash-app to a unique name
$ git add . # add all files to git
$ git commit -m 'Initial app boilerplate'
$ git push heroku master # deploy code to heroku
$ heroku ps:scale web=1  # run the app with a 1 heroku "dyno"
```

dashboard_deployment hosted with ❤ by GitHub view raw

Code Block 44: Initialize Heroku and Deploy

You should be able to view your app at `https://my-dash-app.herokuapp.com` (changing `my-dash-app` to the name of your app).

- **Step 5**: Update the code and redeploy. When you modify any element of your app, you will need to add the changes to git and push those changes to Heroku. Also, whenever I change the underlying CSV data file, you need to push the new data set to Heroku.

```
git status # view the changes
git add .  # add all the changes
git commit -m 'a description of the changes'
git push heroku master
```

dashboard_deployment_update hosted with ❤ by GitHub view raw

Code Block 45: Pushing changes to Heroku via Git

I should note that if you have issues deploying to Heroku, try deploying a simply app first and test whether you are doing each step correctly. For instance, I found that I needed to add the following to the `index.py` file in order for the Heroku deployment to work: `from app import server`.

Resolving an Issue with Custom.css File and Authentication

In addition, I found that when I added authentication to my app, the app could no longer access my `custom.css` file, so I needed to create a css file on codepen.io and include it in the list of external css files (I am assuming that this is a bug in Dash).

Dash Deployment Server

Plotly also offers a Dash Deployment Server…

> *for easy app deployment in commercial IT environments, get started with support plans and workshops for mission critical projects, or contract our world-class engineering team for custom feature development or proof of concept app development.*

I should note that I have no affiliation with Plotly.

. . .

15. Wrapping up

From start to finish, the project took about two and half weeks. I had not previously used Dash and had limited experience using Plotly. Some of the methods which I employed to build the dashboard included:

- Building minimal parts of the dashboard first. Basically, I do not try to tackle everything at once. I built element at a time, and often by itself, before adding it to the entire dashboard.
- To ensure that I got my python syntax correct, I used a Jupyter notebook with the original data frame and applied each change in a piecewise fashion.
- I would make small incremental changes to the app and then test. Rinse and repeat.
- I developed one section first and got it working and to my liking. Then I would copy and paste this section to the other sections and make appropriate changes.
- I saved deployment for the last step. I tested everything locally.

Some things on my wish list to update the Dashboard include:

- Add weeks to each graph axis, most likely using annotations.
- Have the graph x-axis cross over years, rather than just depict either 2018 or 2019.

- Add a “print PDF” button for the report. I tried to implement one but haven’t been able to get it working yet.
- Add a footer with metadata (data sources, date data was pulled, where it was pulled from, etc.)

Thanks for reading this far. I know it is a little overwhelming, but I essentially broke things down into manageable pieces.

. . .

16. Resources for Dash

Here are some resources to help you further with Dash:

- Dash User Guide
- Udemy Course (I have not taken this online course so I cannot comment on it though.
- Dash Community Forum
- Awesome Dash Resource Guide on Github

Dash Dashboard Data Science Data Visualization Towards Data Science

About Help Legal