Product Search Documentation

## Brandon Takaki, Summer ‘21

###### Annotations:

L123 -> Line 123

# **productsearch.py**

Contains everything about the search applications components from Plotly and Dash. Includes component creation, layout, interaction, and subsequent helper function. Also handles what local address application is run on.

Plotly Line Graphs: <https://plotly.com/python/line-charts/>

Dash components: <https://dash.plotly.com/dash-core-components>

Dash HTML for python abstraction: <https://dash.plotly.com/dash-html-components>

Dash DataTable: <https://dash.plotly.com/datatable>

DataFrame form pandas: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>

# Component Creation and Layout

##### L539 | Application Layout

* Contains text detailed last updated demo
* Tab Dash component used for switching which search is used.
  + Callback function (L553): When tab is switched, entire search layout container is rebuilt for new product.
* Search container: initialized to mixer search.

##### L298 | search\_container(product): Search Container Layout

* Different components will have their attribute **className** set to different column numbers or as a row.
* Number of columns tied to a percentage specified here under Grid (L37): <https://codepen.io/chriddyp/pen/bWLwgP.css>
* Columns can be visualized here: <https://codepen.io/chriddyp/pen/bWLwgP>

L299 | **container\_children**: list to be filled and returned as the children of outer seach container **Div** object.

* The order components are added will dictate the position of their row; first on top, last on bottom.

L46 | **add\_error()**: created confirm dialog Dash component used as an error popup message

L303 | **load\_class\_vars()**: initializes all the class variables of **Product** to fit parameter **product**.

L49 | **add\_header(title)**: Informational Text to go at the top of the window.

L72 | **add\_input(idname, text, unit, hid)**: Creates two input dash components and two text components that can take a user numeral input.

L136 | **add\_buttons()**: create ‘search’ and ‘reset’ button Dash components to trigger product searching or input resetting.

L171 | **add\_settings\_text()**: add literal text that says ‘Settings’ that will be clickable to show/hide graph options.

L192 | **add\_checklist(options, values)**: checklist dash component initialized hidden, but when revealed can change which graphs are displayed.

L212 | **add\_radiobuttons(options, value)**: radioitems dash component changed the sorting method of product table. Not added for some product searches.

L225 | **add\_table(columns, sortby, hiddencolumns, styledataconditional)**: DataTable dash component that will display the searched products that fit the inputted parameters. Wrapped in a loading dash component to indicate when still searching.

L286 | **add\_figures()**: graphical figures will be added as children of the Div container with id=’graphs’.

# Component Callback Functions – Interactivity

Dash Callbacks: <https://dash.plotly.com/basic-callbacks>

L825 | **settings\_toggle(clicks)**: Clicking settings text will show/hide graph selecting checklist.

L843 | **reset\_inputs(clicks):** Clicking reset button clears all inputs.

L861 | **update\_graph(…)**:

* Inputs: triggered by selecting products in product table or changing checklist values
* User inputs to determine x ranges of graphs in **inputs**
* L662 | **manage\_load(…)**:
  + based off of the selected products, create product objects and remove unneeded ones
* L890 | desired graphs are generated with **generate\_figures(…)**
* Output: all generated graphs are outputted to be displayed into the id=’graphs’ Div object

L982 | **table\_interact(…)**:

* Search button clicked:
  + inputs are manipulated to be useable values included handling missing bounds
  + L1006 | parts that fit inputs are searched for with **search\_products(…)**
  + searched objects are put in global **SEARCHED\_PRODUCT\_OBJECTS** so currently searched for products are always known
  + Data to be displayed in product table are generated based of of the searched products
  + Output: product data dislayed on table
* sort radiobuttons switched:
  + The table is resorted based off of radiobutton value and any new data that needs to go on table is calculated.
* settings checklist changed:
  + P1dB column of datatable is shown/hidden based on checklist selection
  + only in mixer search

# Graph Figure Generation

L704 | **generate\_figures(class\_name, active\_products, input)**:

* **graph\_figures**: list of graphical figures to be returned and displayed
* loop though graphs that need figures in **ACTIVE\_GRAPHS**
  + figure for graph generated from **create\_graphs(…)**
  + figured added to **graph\_figures**
* **graph\_figures** returned

L722 | **create\_graph(class\_name, active\_products, graph\_type, input)**:

* L724 | figure is modified by color, font, and toolbar options removed.
* for all the products selected to be graphed:
  + for all lines to be ploted for the graph of this product:
    - line is generated from the data of the product, graph labels of the product.
* L758 | figure modified so all the graph labels are correctly fonted and colored.
* L797 | **minmaxy(…)** used to determine the scaling of the graph in the y axis based on the line data within the bounds of the x range to be plotted.
* L799 | figured x and y axis range bounded

# Product Table Functions

Read after looking at the Product class and subclasses

**##\_table\_data(…)**

* performs functions for all searched products.
* retrieves product speficications and information, adds other information and data that will go onto the product table.
* for mixers, power dividers, and baluns, their tables involve minimum, median, and maximum values. These pull the needed values based on the search frequency range. Mixers will pull different graph values based on what the table is current sorting by.
* medians will have a value and a string value, strings for display and values as a hidden column for sorting.
* for graphs with multiple lines, the second min/med/max values will be shown inside parenthesis which is why they are stored as strings.

# **Product**

Root parent class for all product classes for objects that will hold relevant product data and labels

### Class Attributes:

Initialized/changed manually in **productsearch.py** and contains information relevant to the product category as a whole.

**graph\_options**

* list of string representing titles of graphs that are available to be viewed for the resepective product category

**graph\_labels**

* dictionary: key – graph title, value – dictionary
* inner dictionary: key – what label value is for, value – string value for respective label
* labels are for x axis, y axis, x unit, y unit

**products**

* the product category module’s function **load\_specs()** is called
* reads from an excel that has a list of all the searchable products with relevant product specifications and information.
* **products** is a dictionary containing all this information
* dictionary: key – product name/id, value – dictionary of specs and info

**data­\_sheets**

* dictionary for the class to store already read data so data doesn’t need to be pulled from excels multiple times.
* dictionary: key – execl file path, value – dictionary of excel dataframe
* excel dictionary: key – ‘Mapping’, value – dataframe

### Object Attributes:

Inherited by every subclass

**name** - product name

**datasheet**  - web link to the product online datasheet

**color** – color to be used for all of this product’s graph lines plotted

**data**

* dictionary containing all loaded graph data.
* dictionary: key – graph title, value – dictionary of line data
* line data dict: key – line name/label, value – dictionary of axis values
* axis values dict: key – ‘xdata’ or ‘ydata’, value – list of numerical data values

**linekeys**

* dictionary containing the names of all the lines plotted
* dictionary: key – graph title, value – list of line names/labels

### Object Methods:

Inherited by every subclass

**get\_col\_data()** – Returns the specs and info about the product from **products**

**getdata(graph, d=None)**

* returns graph data
* d is line name/label of data wished to be retrieved
* if d is None return all line data of graph

**getlinekeys(graph)**

* return all list of line names/labels for the graph

**set\_color()**

* set what color this products lines should be

**getystats(xlow, xhigh, graph)**

* calculate minimum, median, and maximum values of the graph data within a specific x range.
* If the graph has two lines present the secondary line also has it’s min, med, max values calculated

### Class Functions

**get\_cell\_parameters(df, row, col)**

* from the given dataframe, goes to the specific row and column location and reads the line parameter data.
* reads: line label, x sheet, x column, x min cell, x max cell, y sheet, y column, y min cell, y max cell

**getcelldata(graph index, excel path, x sheet, x column, x min, x max**

**y sheet, y column, y min, y max)**

* If the sheet has already been read from the excel the dataframe is retrieved from **data\_sheets**, otherwise it is read with **pandas.read\_excel()**
* the relevant cells are pulled from the dataframe and fitted into a dictionary
* dict: key: ‘xdata’ or ‘ydata’, value – list of data values

# **Mixer/Amplifier**

Subclass of Product. Inherits all object attributes and methods.

### Object Attributes:

**spreadsheet** - file location of excel that contains the product’s graph data.

### Object Methods:

**load\_graph\_data(graph)** – loads data for the give graph so it can be accessable from **data**

**loaddata(graphindex, row, col)**

* uses the parameters from the excel’s mapping sheet to pull data and populate the **data** and **linekeys** attributes.

**load\_class\_vars()**

* Set the class attributes to be representative of being for the MIXER/AMPLIFIER search.

### Class Functions

**load\_specs()**

* from the product’s specification excel, reads the list of all the searchable products and relevant specifications and information.

# **Passive**

Subclass of Product. Inherits all object attributes and methods.

### Object Attributes:

**filepath** – file path of product’s snp data file

**touchstone**

* object that has all of the data from the snp file and is easy to read from

**touchstone\_data**

* a dictionary containing all of the snp data

### Object Methods

**get\_frequency\_data()** – converts data to GHz from Hz

# **PowerDivder/Coupler/Balun**

Subclass of Passive. Inherits all object attributes and methods of both Product and Passive.

### Object Attributes:

Power Divider: **commonport, outport1, outport2**

Coupler: **coupledport, inport, outport**

Balun: **commonport, outport0, outport180**

### Object Methods:

**set\_port\_config()**

* using the snp file comments/header, determine the port configuration and initialize the port attributes

**load\_class\_vars()**

* Set the class attributes to be representative of being for the POWERDIVIDER/COUPLER/BALUN search

**get\_####\_data()**

* the #### represents a graph.
* every graph has different means of pulling from **touchstone\_data**. One method could straight pull a single column from a single key. Another might use multiple values from different columns to calculate the data at each point.
* get data methods are incorrect/incomplete for Balun

**handle\_outliers(xdata, ydata)**

* only present in PowerDivider get data methods
* There were many graphs with singular points that were huge outliers. They caused graph to significanlty zoom out so the details of the actual line could not be observed.
* This method removes those outlier points.

### Class Functions

**load\_specs()**

* from the product’s specification excel, reads the list of all the searchable products and relevant specifications and information.

# **skrf.py**

Used for reading SNP files. This module houses the **Touchstone** class, attributes, and methods used in **Passive**, and its subclasses. Normally, you could install the dependancy library ‘scikit-rf’, and then just **import skrf as rf**, except Docker (see deployment section) was having issues connecting that skrf and scikit-rf were the same modules even though they had different names. Could never solve the issue so I just copied the scikit-rf code needed and made my own skrf module.

origional library code:

* <https://scikit-rf.readthedocs.io/en/latest/_modules/skrf/io/touchstone.html>

introduction to library use:

* <https://scikit-rf.readthedocs.io/en/latest/tutorials/Introduction.html#>

# **Running Locally for Testing/Development**

Just run, but make sure that the **run\_server** function (L1395) has default params.

# **Heroku**

Use heroku if you need to share the web application to someone for demo use.

<https://dash.plotly.com/deployment>

The free server it deploys the application on is pretty slow. Actions might not register and may require repeated actions for it to register. But this is useful for showing other people since they can just go to a website to see the application.

The current demo sight I have for the application is:

However you will have to make your own since you will use your own heroku account.