vois Documentation

Release 1.0

Davide De Marchi

Apr 20, 2024

CONTENTS

vois

vois library

CHAPTER

ONE

INTRODUCTION

VOIS is a python library to simplify the creation of Voilà dashboards. It contains functions and classes that allow for fast development of data analytics dashboards containing charts (using Plotly python library), datatables from Pandas DataFrames, SVG plots, and interactive maps.

The library contains an example dashboard that is built step by step using the functions and classes of the vois library. This example dashboard displays, in various modes, data on energy consumption of the european countries. Data used by the example dashboard was downloaded from EUROSTAT.

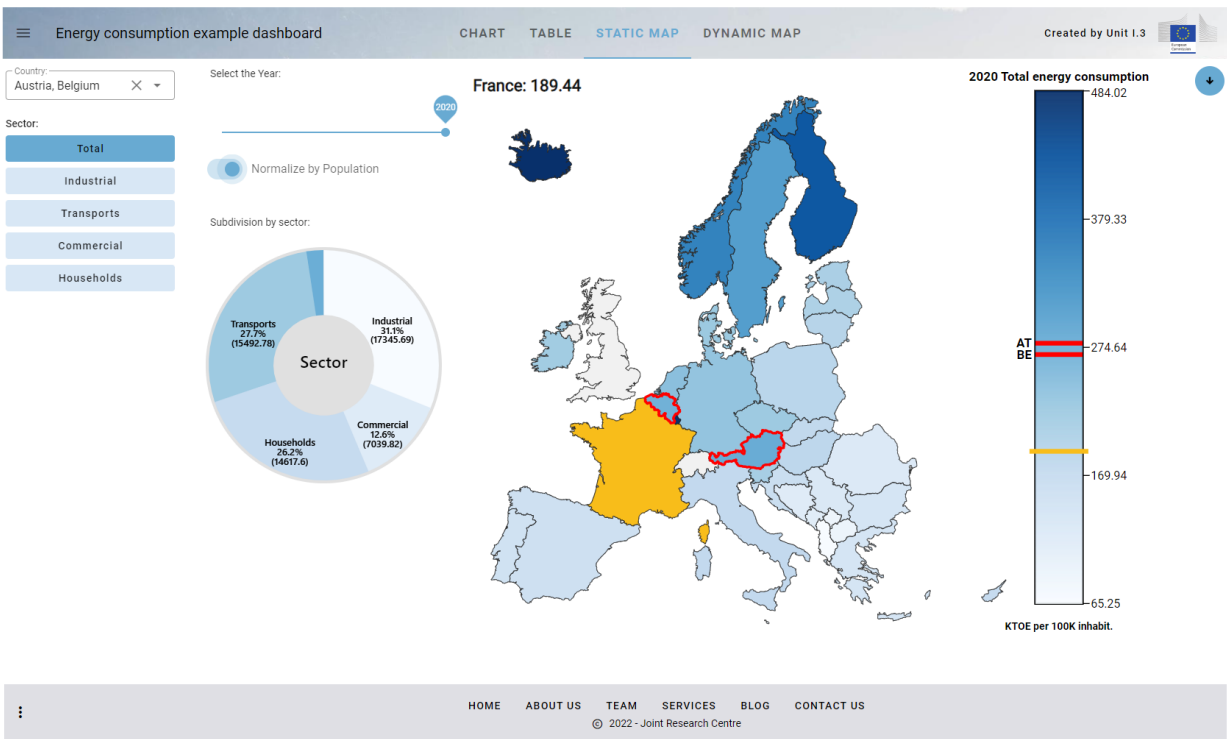


Fig. 1.1: Energy consumption example dashboard

1.1 Widgets libraries

Widgets are the part of a GUI that allows the user to interface with the application. Widgets can make Jupyter notebooks look lively and interactive. Widgets are elements like buttons, drop-down list, slider, etc.

Widgets allow users to interact with the notebook, manipulate output according to the selection of widget and controlling events. It can be used to record the user's input and can be implemented easily in a Jupyter notebook.

Ipywidgets is an open-source python library that is used to generate different types of widgets and implement them in a Jupyter notebook. It is easy to use and provides a variety of interactive widgets. It is the "traditional" way to create GUI elements inside a Jupyter notebook, and is the first choice when a dashboard has to be built.

Fig. 1.2: Some of the standard ipywidgets GUI elements

The look&feel of the ipywidgets, however, cannot be customized, and when there is the need to create fancy looking applications in Jupyter or Voilà, more modern widgets library come to play their role.

For this reason **vois library** uses primarily Ipyvuetify as the widgets base library.

Ipyvuetify is a widget library for making modern looking GUI's in Jupyter notebooks and dashboards (Voilà). It's based on the Google material design philosophy best known from the Android user interface. A large set of widgets is provided with many widgets having multiple variants.

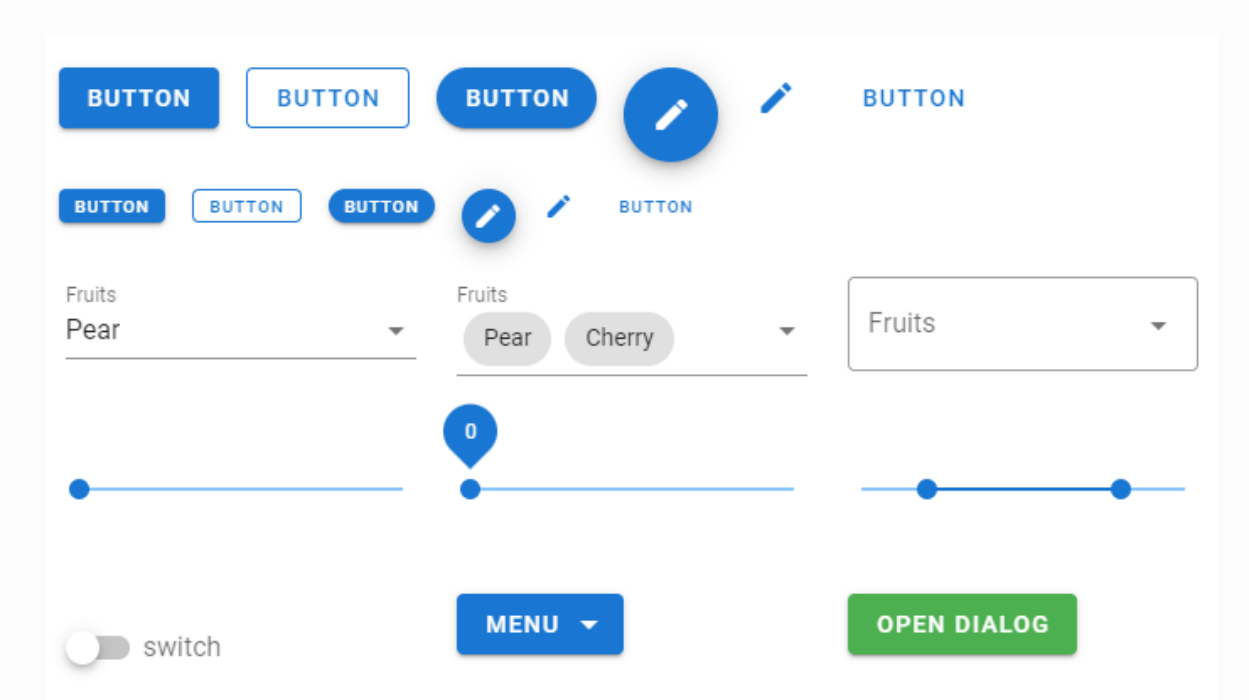


Fig. 1.3: Ipyvuetify sample widgets

To explore which widgets are available in ipyvuetify and how to use them it is useful to view the Vuetifyjs documentation where a very datailed description of the components can be found. You can browse examples on the left-hand side and see the source code by clicking on '< >' on the top right-hand side of the example. By reading the Usage section of ipyvuetify documentation you will be able to translate the examples to ipyvuetify.

When comparing ipyvuetify to ipywidgets, the standard widget library of Jupyter, ipyvuetify has a lot more widgets which are also more customizable and composable (in term of colors, shapes, functions, etc.) at the expense of a bit

more verbosity in the source code. Using ipywuetify requires more effort compared to ipywidgets. For this reason, one of the main scope of this vois library is to simplify the use of ipyvuetify widget, so that the development of fancy Voilà dashboards becomes easier.

1.2 Packages

The vois library is grouped in these packages:

General package	Contains modules that define utilities functions and classes of general use (geojson, maps, svg, etc.)
Vuetify package	Contains modules that define classes to simplify the creation of GUI elements using ipyvuetify widgets

vois library

CHAPTER

TWO

TUTORIAL

2.1 Energy consumption dashboard

This tutorial will provide a step by step description of the construction of a dashboard to analyse and display data on Energy Consumption in Europe. The steps of the example dashboard are available as notebooks inside the **examples/EnergyConsumption** subfolder. The data for this example has been downloaded from EUROSTAT web site, in particular:

File ten00124_linear.csv: Energy consumption data

```
import pandas as pd
df = pd.read_csv('./ten00124_linear.csv')
df
```

	DATAFLOW	LAST UPDATE	freq	nrg_bal	siec	unit	geo	TIME_PERIOD	OBS_VALUE	OBS_FLAG
0	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2009	1840.737	NaN
1	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2010	1897.918	NaN
2	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2011	1953.468	NaN
3	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2012	1790.623	NaN
4	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2013	1961.871	NaN
...	...	...	...	...	...	...	...	...	...	...
2460	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_TRA_E	TOTAL	KTOE	XK	2016	395.981	NaN
2461	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_TRA_E	TOTAL	KTOE	XK	2017	410.191	NaN
2462	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_TRA_E	TOTAL	KTOE	XK	2018	429.875	NaN
2463	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_TRA_E	TOTAL	KTOE	XK	2019	428.184	NaN
2464	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_TRA_E	TOTAL	KTOE	XK	2020	421.572	NaN

2465 rows × 10 columns

Fig. 2.1: Input data for energy consumption in european countries

File tps00001_tabular.tsv: Population data

```
import pandas as pd
df = pd.read_csv('./tps00001_tabular.tsv', delimiter='\t')
df
```

	freq,indic_de,geo\TIME_PERIOD	2010	2011	2012	2013	2014	2015	2
0	A,JAN,AD	84082	78115 b	78115	76246	:	:	71
1	A,JAN,AL	2918674	2907361	2903008	2897770	2892394	2885796	2875
2	A,JAN,AM	3249482	3262650	3274285	3026878	3017079	3010598	2998
3	A,JAN,AT	8351643	8375164	8408121	8451860	8507786	8584926	8700
4	A,JAN,AZ	8997586	9111078	9235085	9356483	9477119	9593038	9705
5	A,JAN,BA	3844046	3843183	3839265	3835645	3830911	3825334	3515
6	A,JAN,BE	10839905	11000638 b	11075889 b	11137974	11180840	11237274	11311
7	A,JAN,BG	7421766	7369431	7327224	7284552	7245677	7202198	7153
8	A,JAN,BY	9480178	9481193	9465150	9463840	9468154	9480868 e	9498
9	A,JAN,CH	7785806	7870134 b	7954662	8039060	8139631	8237666	8327
10	A,JAN,CY	819140	839751	862011	865878	858000	847008	848
11	A,JAN,CZ	10462088	10486731	10505445	10516125	10512419	10538275	10553
12	A,JAN,DE	81802257	80222065 b	80327900	80523746	80767463	81197537	82175
13	A,JAN,DK	5534738	5560628	5580516	5602628	5627235	5659715	5707
14	A,JAN,EA18	332124448	331520001 b	332285283 b	333073061	334820880 b	335640859 b	336899
15	A,JAN,EA19	335266424	334572589 b	335288924 b	336044966	337764352 b	338562121 b	339787
16	A,JAN,EE	1333290	1329660	1325217	1320174	1315819	1314870 b	1315
17	A,JAN,EL	11119289	11123392	11086406	11003615	10926807	10858018	10783

Fig. 2.2: Input data for the population of european countries

2.1.1 Dashboard steps

Scope of the example dashboard on "Energy Consumption in Europe" is to demonstrate a real example of a dashboard built using the **vois library**.

The dashboard loads the input data from CSV files stored in the same folder of the notebooks using Pandas and displays it in four modes: a chart, a table, a static map and a dynamic map. By "static map" it is meant that a drawing representing a map is displayed, without the possibility to do pan or zoom operations. This display mode is obtained using an SVG drawing. On the contrary, the "dynamic map" is based on a mapping widget that allows users to pan, zoom and click on the geographic features.

Some widgets are added in the various steps to select countries and/or energy sectors and update the views accordingly.

The dashboard is created in 10 (+1) steps, here listed:

Step	Description	Link to dashboard	Source code
1	Creation of the dashboard structure using the `app.app` class	Link 1	Source 1
2	Use Pandas to read the input CSVs and display the DataFrame as a table	Link 2	Source 2
3	Add the filtering controls to the dashboard to select countries and sector	Link 3	Source 3
4	Add the Plotly Bar Chart View	Link 4	Source 4
5	Add the SVG Static Map View	Link 5	Source 5
6	Add the Dynamic Map View	Link 6	Source 6
7	Add the functions for downloading chart, table and map	Link 7	Source 7
8	Manage the parameters passsed in the URL and open external URLs	Link 8	Source 8
9	Add an interactive AnimatedPieChart in SVG to select the consumption sector	Link 9	Source 9
Final	Add minipanel to footer bar and function to generate and download report in docx format	Link Final	Source final
Final dark	A "dark" version of the final dashboard	Link Final dark	Source final dark

2.1.1.1 Step 1: Creation of the dashboard structure

The initial step of the dashboard creates the graphical structure of the application using the `app.app` class.

The first cell contains the customization of the settings that influence the appearance of all the graphical widgets. Here the dark_mode flag is set to False and the main colors of the widgets are defined (settings.color_first and settings.color_second), by selecting two shades of blue/cyan. Then the flag that defines if the button shape is round is set to False (so the buttons will have a 'squared' shape in this dashboard). See `settings` for reference:

```
from vois.vuetify import settings
settings.dark_mode      = False
settings.color_first    = '#68aad2'
settings.color_second   = '#d8e7f5'
settings.button_rounded = False
```

Then the import of the libraries used in this version of the dashboard (ipywidgets and ipyvuetify):

```
from ipywidgets import widgets, Layout, HTML
from IPython.display import display
import ipyvuetify as v
```

Followed by the import of the app module of the vois library:

```
from vois.vuetify import app
```

Ths second cell creates a global variable called **g_app** containing an instance of the *app.app* class. All the aspect of the app appearance are defined in the call to **app.app**(): Title of the dashboard, list of tabs to display in the title bar, credits to show on the right of the title bar, content of the side panel and of the footer bar. To have a list of all the parameters, please consult the page on the *app.app*.

Four callback functions are defined that simply display a **snackbar** message to the user when one of the buttons of the app interface is clicked (see *snackbar.snackbar* module). This messages by default remain visible for 10 seconds and then automatically disappear. The user can click on the 'Close' button of the snackbar to close it before the 10 seconds timeout:

```
# Click on a tab of the title
def on_click_tab(arg):
    g_app.snackbar(arg)

# Click on the credits text
def on_click_credits():
    g_app.snackbar('Credits')

# Click on the logo
def on_click_logo():
    g_app.snackbar('LOGO')

# Click on the footer buttons
def on_click_footer(arg):
    g_app.snackbar(arg)

g_app = app.app(title='Energy consumption example dashboard',
                titlecredits='Created by Unit I.3',
                titlewidth='60%',
                footercolor='#dfdfe4',
                footercredits='Data',
                footercreditstooltip='Eurostat - European Commission',
                footercreditsurl='https://ec.europa.eu/eurostat/data/database',
                titletabs=['Chart', 'Table', 'Static Map', 'Dynamic Map'],
                titletabsstile='font-weight: 700; font-size: 17px;',
                dark=False,
                backgroundimageurl='https://picsum.photos/id/293/1920/1080',
                sidepaneltitle='Help',
                sidepaneltext="Lorem ipsum dolor sit amet, consectetur adipiscing elit,␣
↪sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim␣
↪veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo␣
↪consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum␣
↪dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt␣
↪in culpa qui officia deserunt mollit anim id est laborum.",
```

(continues on next page)

(continued from previous page)

```
                sidepanelcontent=[v.Icon(class_='pa-0 ma-0 ml-2', children=['mdi-help
↪'])],
                onclicktab=on_click_tab,
                onclickcredits=on_click_credits,
                onclicklogo=on_click_logo,
                onclickfooter=on_click_footer)

g_app.show()
```

After the two cells are executed, this is the appearance of the dashboard:

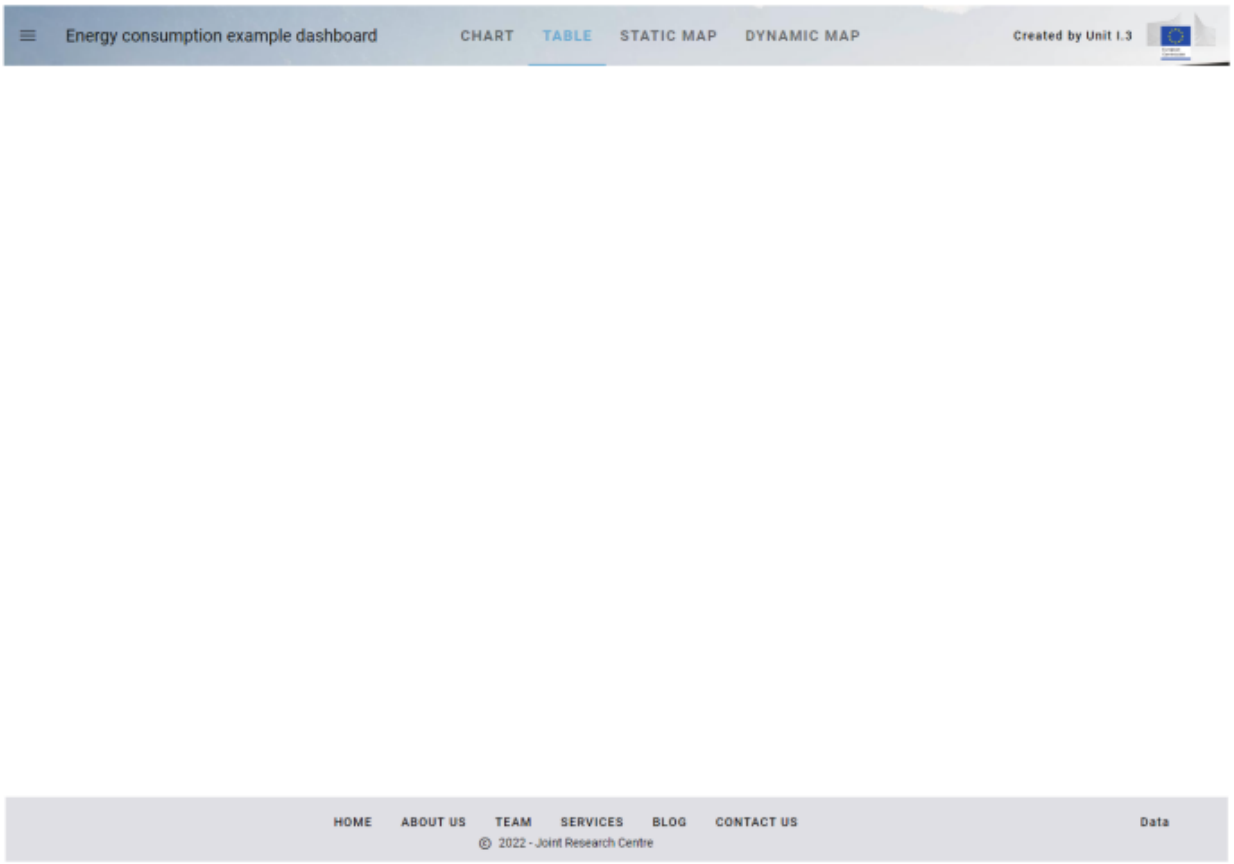


Fig. 2.3: Appearance of the dashboard on step 1

The title bar and the footer bar are displayed, while the central part of the dashboard is empty. In the following steps this part will be filled with visualisations of the input datasets and widgets controls to allow for selections and filtering of input data.

The side panel of the application is already working: it can be opened by clicking on the top-left icon in the title bar.

The button 'Data' displayed in the bottom-right corner of the footer bar is working and opens the EUROSTAT web page in another tab of the browser.

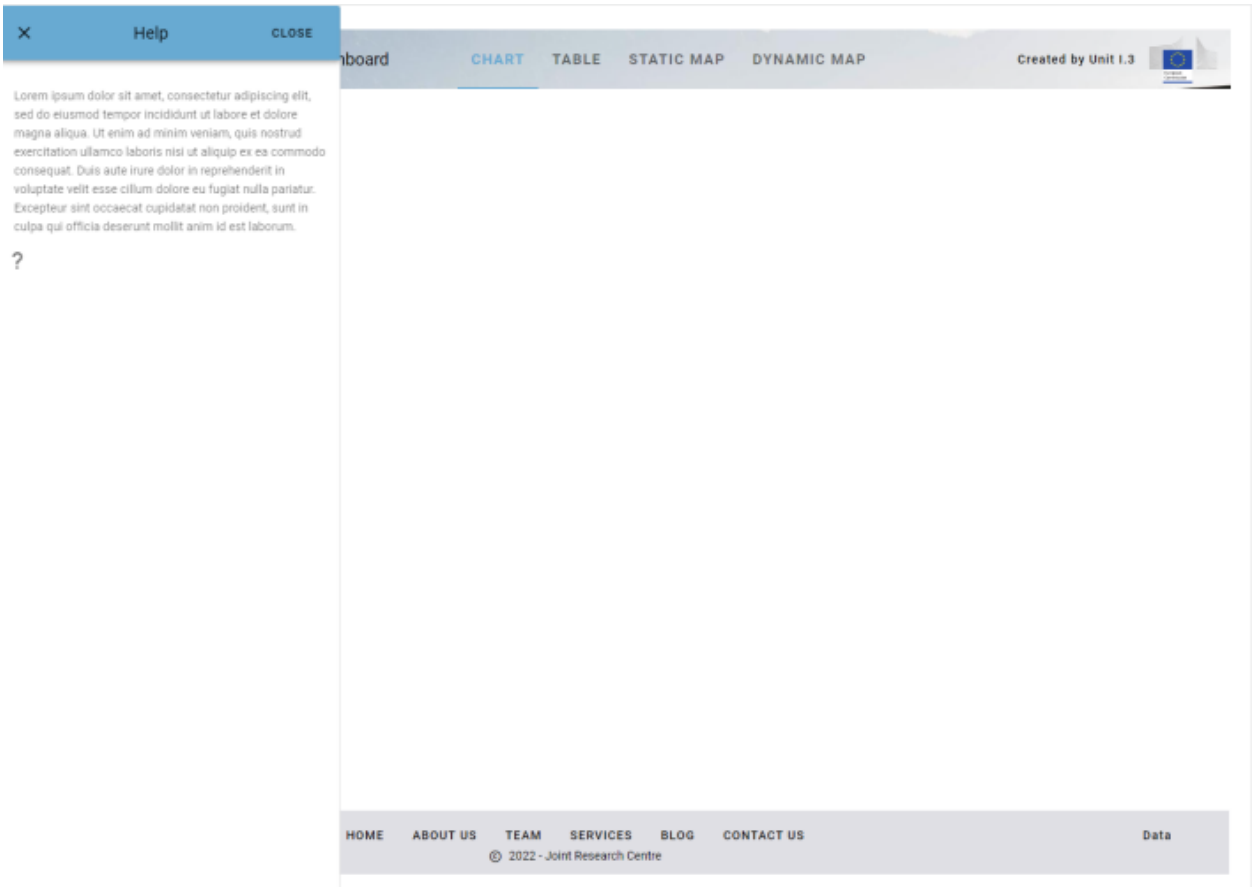


Fig. 2.4: Opening the sidePanel to show Help content

2.1.1.2 Step 2: Use Pandas to read the input CSVs and display the DataFrame as a table

The second step uses the Pandas library to read the input CSVs and display the DataFrame as a table.

In the first cell, the import of the Pandas library is added:

```
import pandas as pd
```

Followed by the import of the datatable module of the vois library:

```
from vois.vuetify import app, datatable
```

Then the import of the local module EnergyConsumption.py is added. This module contains the python code for loading the input data:

```
import EnergyConsumption
```

Here is the code code of the EnergyConsumption.py module:

```
# Loads input data and returns a pandas dataframe
def loadData():
    # Load energy data
    df = pd.read_csv('./ten00124_linear.csv')

    # Remove columns that are not useful
    df.drop(['OBS_FLAG'], axis=1, inplace=True)

    # Assign Country to df
    df['Country'] = df['geo'].map(code2name)

    # Load population data
    dfpop = pd.read_csv('./tps00001_tabular.tsv', delimiter='\t')

    # Dict key=Code2char  value=Population
    countrypop = {}

    dfpop.rename(columns={'freq,indic_de,geo\TIME_PERIOD': 'geo'}, inplace=True)
    columns = dfpop.columns
    for index, row in dfpop.iterrows():
        geo = row['geo'].split(',')[2]
        for col in reversed(columns):
            if len(row[col]) > 2:
                pop = int(row[col].split(' ')[0])
                if pop > 0:
                    countrypop[geo] = pop
                    break

    # Assign population to df
    df['Population2021'] = df['geo'].map(countrypop)

    return df
```

After reading the two datasets, the loadData function adds a new column named 'Population2021' to the Pandas DataFrame by taking the rightmost (i.e. more recent) column for each country that contains valid data. In order to purge unnecessary data, some of the columns of the input dataset that contain not useful data (like the 'siec' and the

'unit' columns) could be removed from the Pandas DataFrame. The DataFrame returned by the call to the EnergyConsumption.loadData function is used in all the subsequent steps of the example dashboard and is stored in the global variable named **g_df**:

```
g_df = EnergyConsumption.loadData()
```

A third cell is added to the notebook, before the creation of the **g_app** instance, to display the global dataframe **g_df** inside the Output content of the app instance (see *app.app.outcontent*):

```
def displayDatatable():
    g_app.outcontent.clear_output(wait=True)
    d = datatable.datatable(data=g_df, height='800px')
    with g_app.outcontent:
        display(d)
```

Here the **g_app.outcontent** is the main Output widget that covers all the empty space between the title bar and the footer bar of the app instance. Please note the usage of the *datatable.datatable* class to display the Pandas DataFrame as a datatable.

Note: The **clear_output** method of the ipywidgets.Output class takes an optional parameter named **wait** of type boolean. If this parameter is set to True, the Output widget is not cleared immediately, but only when other content is displayed in the widget. This allows for smooth transitions among different contents and avoids the effect of 'flickering'. For additional info, see ipywidgets documentation: Output widgets: leveraging Jupyter's display system

In the last cell of the notebook, the displayDatatable() function is called, just after the creation of the **g_app** instance.

2.1.1.3 Step 3: Add the filtering controls to the dashboard to select countries and sector

The third step adds the filtering controls to the dashboard to select countries and sector, so that the Pandas DataFrame can be browsed per country and per energy sector.

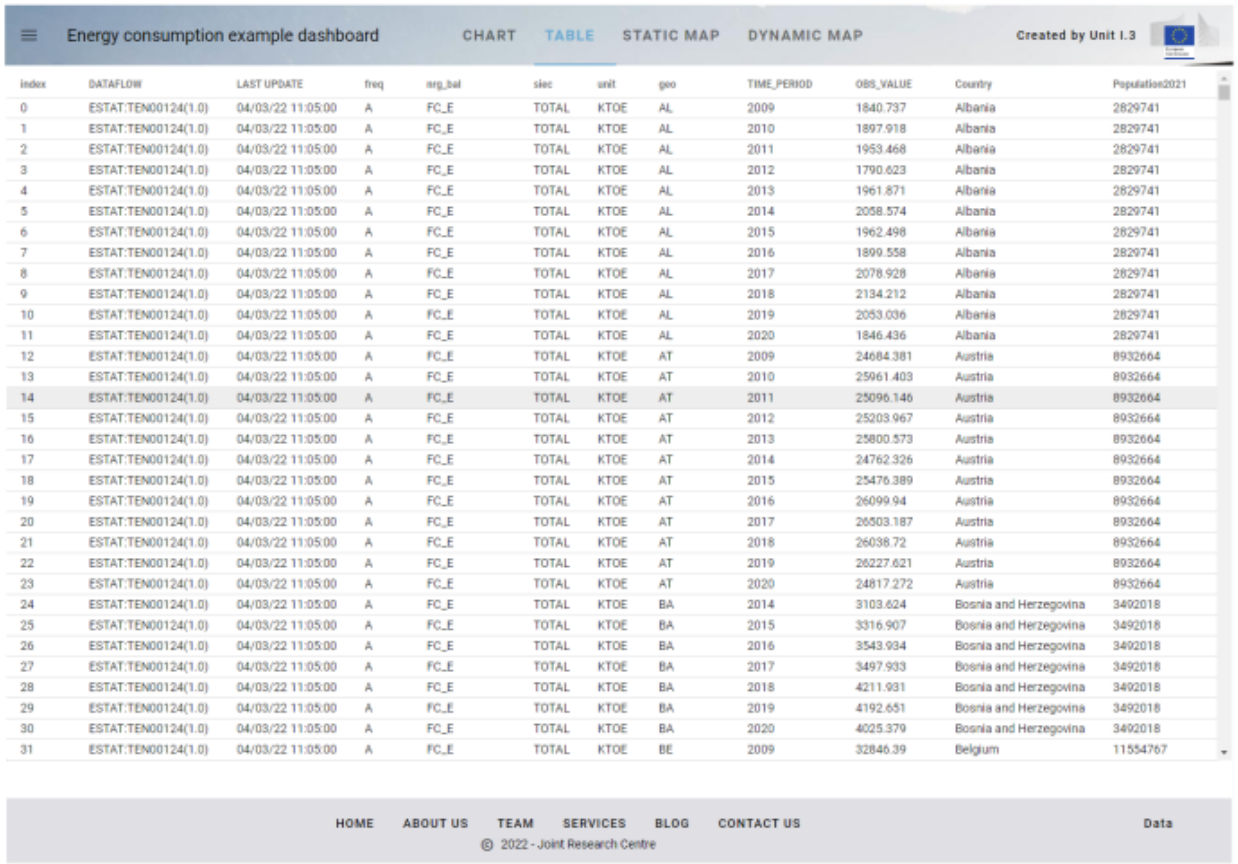
In the first cell of the notebook, the import of selectMultiple, label and toggle widgets is added:

```
from vois.vuetify import app, selectMultiple, label, datatable, toggle
```

A cell is added in second position in the notebook. It contains code to create the ipywidgets.Output widgets that subdivide the space of the dashboard in parts for specific usage:

```
#border = '1px solid lightgrey'
border = 'none'

# Dimensioning
widthinpx     = 260
widthControls = '%dpx' % widthinpx

heightinpx = 830
height     = '%dpx' % heightinpx

height_net = '%dpx' % (heightinpx-10)

outControls = widgets.Output(layout=Layout(width=widthControls, min_width=widthControls,␣
```

(continues on next page)



index	DATAFLOW	LAST UPDATE	freq	nrg_bal	siec	unit	geo	TIME_PERIOD	OBS_VALUE	Country	Population2021
0	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2009	1840.737	Albania	2829741
1	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2010	1897.918	Albania	2829741
2	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2011	1953.468	Albania	2829741
3	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2012	1790.623	Albania	2829741
4	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2013	1961.871	Albania	2829741
5	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2014	2058.574	Albania	2829741
6	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2015	1962.498	Albania	2829741
7	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2016	1899.558	Albania	2829741
8	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2017	2078.928	Albania	2829741
9	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2018	2134.212	Albania	2829741
10	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2019	2053.036	Albania	2829741
11	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AL	2020	1846.436	Albania	2829741
12	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2009	24684.381	Austria	8932664
13	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2010	25961.403	Austria	8932664
14	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2011	25096.146	Austria	8932664
15	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2012	25203.967	Austria	8932664
16	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2013	25800.573	Austria	8932664
17	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2014	24762.326	Austria	8932664
18	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2015	25476.389	Austria	8932664
19	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2016	26099.94	Austria	8932664
20	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2017	26503.187	Austria	8932664
21	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2018	26038.72	Austria	8932664
22	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2019	26227.621	Austria	8932664
23	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	AT	2020	24817.272	Austria	8932664
24	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2014	3103.624	Bosnia and Herzegovina	3492018
25	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2015	3316.907	Bosnia and Herzegovina	3492018
26	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2016	3543.934	Bosnia and Herzegovina	3492018
27	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2017	3497.933	Bosnia and Herzegovina	3492018
28	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2018	4211.931	Bosnia and Herzegovina	3492018
29	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2019	4192.651	Bosnia and Herzegovina	3492018
30	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BA	2020	4025.379	Bosnia and Herzegovina	3492018
31	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	BE	2009	32846.39	Belgium	11554767

Fig. 2.5: Appearance of the dashboard on step 2

(continued from previous page)

```
↪height=height, border=border))
outDisplay  = widgets.Output(layout=Layout(width='90%', height=height, border=border))
```

In the code, some dimensioning in width and height is defined (expressed in pixels), and two Output widgets are created. The first is named **outControls** and will host some widgets to select the country and the energy sector that originates the consumption (industrial, commercial, etc.). The second Output widget is called **outDisplay** and will contain the visualisation defined by the user acting on the controls.

Interesting to note is the usage of the width in pixels for outControls and the with in percentage for the outDisplay. Combining it with the layout.min_width setting, this allows for having a outControls widget of fixed horizontal dimension of 260 pixels, while the outDisplay will occupy the remaining part of the page, whatever is the total dimension of the screen (fullHD, table, smartphone, etc.).

Also interesting is the first line of the cell where the variable **border** is defined to be 1 pixel and grey. If this line is uncommented, and the following one is commented, the border of the two Output widgets will become visible and this can be useful during the development of the dashboard or for debugging purpose to check that their dimensions behave correctly. This can be achieved also adding a new cell in the notebook and executing these commands:

```
outControls.layout.border = '1px solid grey'
outDisplay.layout.border = '1px solid grey'
```

The cells added in the central part of the notebook will fill the two Output widget with the content. The display of these widgets inside the **g_app.outcontent** is done by this line added in the last cell of the notebook, just after the creation of the app.app instance:

```
with g_app.outcontent:
    display(widgets.HBox([outControls,outDisplay]))
```

This code displays the two Output widget using the ipywidgets.HBox method that serves to display two or more widgets side by side in the same horizontal line (see ipywidgets documentation: Container/Layout widgets)

Following the cell that loads the input data, a new cell is added that defines some global variables:

```
g_countries  = []    # Selected countries codes
g_dtfiltered = None  # Filtered dataframe
g_currentgeo = ''    # list of comma-separated names of the selected countries
g_sector     = 'FC_E'
g_units      = 'Thousand tonnes of oil equivalent'
g_year       = g_df['TIME_PERIOD'].max()

# Ordered list of sectors
g_sectors = ['FC_E', 'FC_IND_E', 'FC_TRA_E', 'FC_OTH_CP_E', 'FC_OTH_HH_E']

# Short names of the sectors
g_sectorTitle = {
                 'FC_E':        'Total',
                 'FC_IND_E':    'Industrial',
                 'FC_TRA_E':    'Transports',
                 'FC_OTH_CP_E': 'Commercial',
                 'FC_OTH_HH_E': 'Households',
                }

# Long names of the sectors
g_sectorName = {
```

(continues on next page)

(continued from previous page)

```
                 'FC_E':        'Total energy consumption',
                 'FC_IND_E':    'Industrial energy consumption',
                 'FC_TRA_E':    'Transports energy consumption',
                 'FC_OTH_CP_E': 'Commercial energy consumption',
                 'FC_OTH_HH_E': 'Households energy consumption',
                }
```

In particular **g_sector** is a list of the codes of the energy sectors, as they appear in the columns of the input Pandas DataFrame, while **g_sectorTitle** and **g_sectorName** are two dictionary to retrieve the short and long name of a sector given its code.

The following cell creates some widgets to filter the input data. A first widget is a *selectMultiple.selectMultiple* instance to select one or more countries. When no country is selected, the total data for Europe27 will be displayed, while when a valid selection is done, only the rows of the **g_df** Pandas Dataframe related to the selected countries will be displayed. This widget is created with the following instruction:

```
selcountry  = selectMultiple.selectMultiple('Country:', EnergyConsumption.eunames,␣
↪width=widthinpx-30,
                                            mapping=countries_mapping, onchange=onchange_
↪country, marginy=2)
```

The values displayed by the selectMultiple widget will be the EUROSTAT Country Names (content of the variable EnergyConsumption.eunames taken from the svgMap module and derived from the EUROSTAT Country Codes) while the mapping with the function **countries_mapping** will convert the name of the countries to their corresponding codes. If name is None the function returns the code for EU27.

The **onchange_country** function is called every time the user makes a selection in the country widget and it changes the value of the global variable **g_countries** to be the list of the codes of the selected countries. Then the UpdateDataframe function is called to update the visualisation:

```
# Selection of a country
def onchange_country():
    global g_countries
    g_countries = selcountry.value
    UpdateDataframe()
    displayDatatable()
```

A second widget is created as a *toggle.toggle* instance to display buttons (displayed in vertical mode) for the energy sectors:

```
selsector = toggle.toggle(0,
                          [g_sectorTitle[x] for x in g_sectors],
                          tooltips=[g_sectorName[x] for x in g_sectors],
                          onchange=onchange_sector,
                          row=False,
                          width=widthinpx-30)
```

Here the title of the sector is displayed in the buttons, while the full name of the sector is used as tooltip when the user hovers one of the buttons.

The **onchange_sector** function is called every time the user makes a selection in the sector widget and it changes the value of the global variable **g_sector** to be the code of the selected sector. Then the UpdateDataframe function is called to update the visualisation:

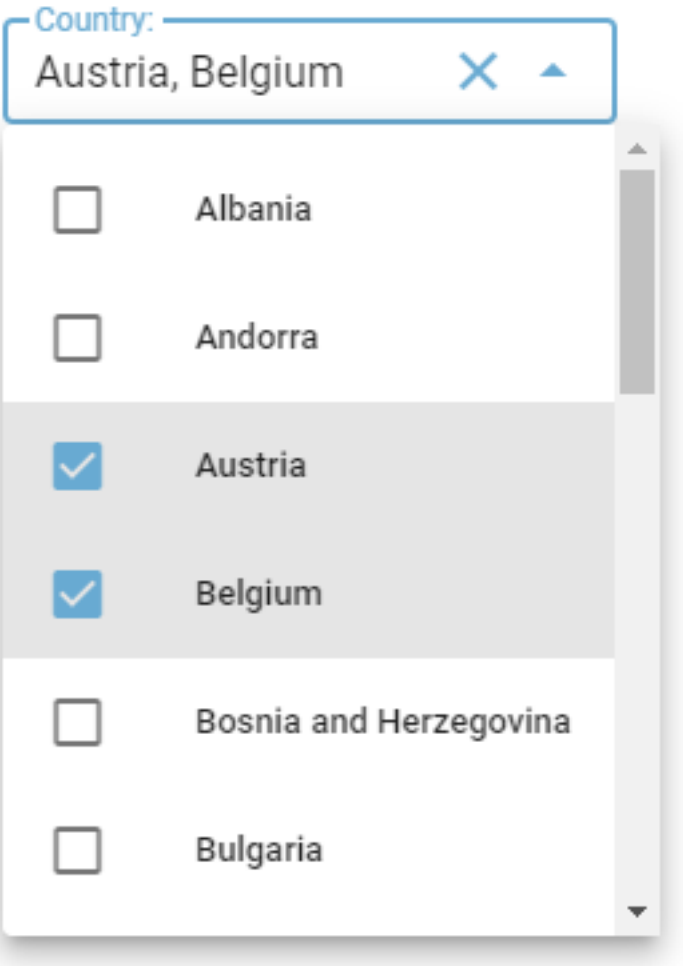


Fig. 2.6: selectMultiple widget for countries

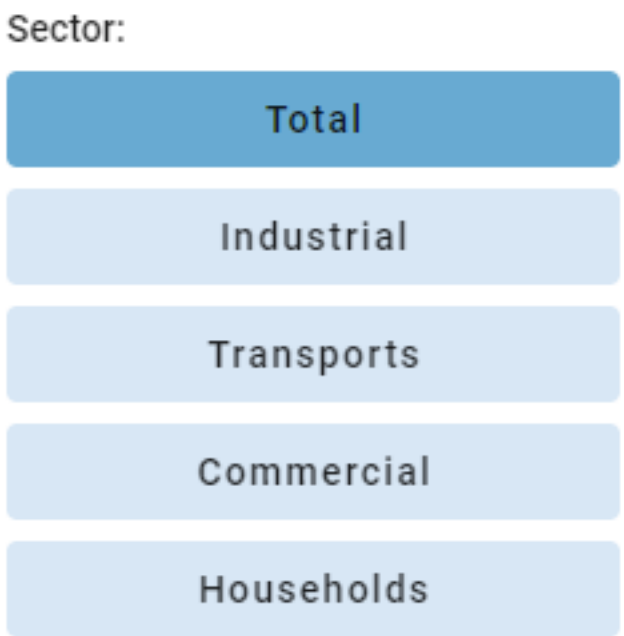


Fig. 2.7: Toggle widget for sectors

```
# Selection of a sector
def onchange_sector(value):
    global g_sector
    g_sector = g_sectors[value]
    UpdateDataframe()
    displayDatatable()
```

The **UpdateDataframe** is responsible for creating a filtered version of the Pandas DataFrame to contain only the rows corresponding to the user selections in the selectMultiple and toggle widgets:

```
# Update the filtered dataframe
def UpdateDataframe():
    global g_dtfiltered, g_currentgeo

    # Filter dataset on country and sector
    if len(g_countries) == 0:
        codes = ['EU27_2020']
        g_currentgeo = 'Europe27'
    else:
        codes = g_countries
        g_currentgeo = ', '.join([EnergyConsumption.code2name[x] for x in g_countries])

    g_dtfiltered = g_df[(g_df['geo'].isin(codes))&(g_df['nrg_bal']==g_sector)].copy()
    g_dtfiltered.rename({'TIME_PERIOD': 'Year', 'OBS_VALUE': g_units}, axis=1,␣
↪inplace=True)
```

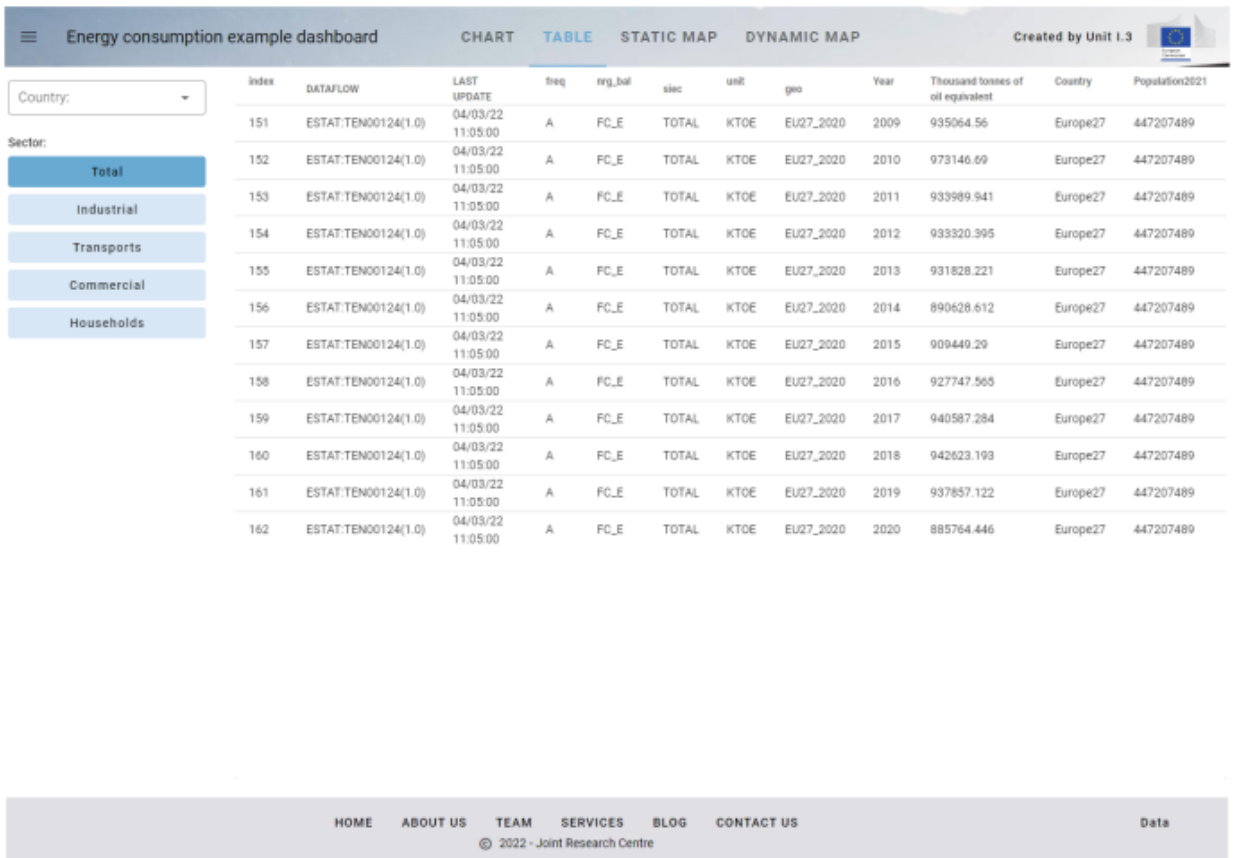
This function updates the global variable **g_dtfiltered** to contain only the rows where the 'geo' column contains one of the codes of the selected countries and where the 'nrg_bal' column corresponds to the selected energy sector.

Finally, the **outControls** widget is filled with the controls:

```
outControls.clear_output()
with outControls:
    display(selcountry.draw())
    display(labelEmpty.draw())
    display(labelSector.draw())
    display(selsector.draw())
```

Since the widget are positioned in vertical one after the other, 4 simple calls to the display method are used. As an altenative one could have used the ipywidgets.VBox layout widget to get the same result. Please note the usage of the **draw()** method which is the method that all the widgets of the vois library implement to return a real instance of the underlining ipyvuetify widget to be displayed.

This is the apperance of the dashboard after the execution of the Step 3 notebook cells:



index	DATAFLOW	LAST UPDATE	freq	nrg_bal	siec	unit	geo	Year	Thousand tonnes of oil equivalent	Country	Population2021
151	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2009	935064.56	Europe27	447207489
152	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2010	973146.69	Europe27	447207489
153	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2011	933989.941	Europe27	447207489
154	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2012	933320.395	Europe27	447207489
155	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2013	931828.221	Europe27	447207489
156	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2014	890628.612	Europe27	447207489
157	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2015	909449.29	Europe27	447207489
158	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2016	927747.565	Europe27	447207489
159	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2017	940587.284	Europe27	447207489
160	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2018	942623.193	Europe27	447207489
161	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2019	937857.122	Europe27	447207489
162	ESTAT:TEN00124(1.0)	04/03/22 11:05:00	A	FC_E	TOTAL	KTOE	EU27_2020	2020	885764.446	Europe27	447207489

Fig. 2.8: Appearance of the dashboard on step 3

2.1.1.4 Step 4: Add the Plotly Bar Chart View

The 4th step adds the Plotly Bar Chart View to the dashboard.

In the first cell of the notebook, the import of the plotly library is added:

```
import plotly.express as px
import plotly.graph_objects as go
```

Here **px** is the Plotly Express version of the Plotly API, while **go** is the Plotly Graph Objects API. Both versions of the library are used in this example.

In the cell containing the global variables, a new variable is added, storing the color sequence selected from those available in the Plotly library (see Discrete Color Sequences in Plotly Express and Built-In Sequential Color scales):

```
# Color sequence to use in the Plotly chart
g_colorsequence = px.colors.sequential.Blues[::-1]
```

A new cell is added to the notebook to define the python function responsible for the update of the **outDisplay** Output widget with a Plotly Bar chart:

```
def displayChart():
    global g_last_fig

    outDisplay.clear_output(wait=False)
    with outDisplay:
        title = g_sectorName[g_sector] + ' for ' + g_currentgeo
        if len(g_countries) <= 1:
            g_last_fig = px.bar(g_dtfiltered, x='Year', y=g_units, color="Country",␣
↪template='plotly_white', text_auto=True, color_discrete_sequence=g_colorsequence)
            g_last_fig.update_xaxes(tickvals=g_dtfiltered['Year'])
        else:
            g_last_fig = go.Figure()
            i = 0
            allyears = set()
            for code in g_countries:
                dfsel = g_dtfiltered[g_dtfiltered['geo']==code]
                years = dfsel['Year'].unique()
                allyears.update(years)
                g_last_fig.add_trace(go.Bar(x=years, y=dfsel[g_units],␣
↪name=EnergyConsumption.code2name[code], textposition="inside", texttemplate="%{y}",␣
↪marker_color=g_colorsequence[i]))
                i += 1
                i = i % len(g_colorsequence)
            g_last_fig.update_layout(barmode='group', template='plotly_white', legend_
↪title='Country', xaxis_title="Year", yaxis_title="Thousand tonnes of oil equivalent")
            g_last_fig.update_xaxes(tickvals=sorted(list(allyears)))

        g_last_fig.update_layout(height=heightinpx-10, margin=dict(t=84, l=0, r=0, b=0),␣
↪title={'text': title})
        g_last_fig.show(config={'displaylogo': False, 'displayModeBar': False})
```

The **displayChart** function updates the **outDisplay** Output widget by clearing its content and replacing it with a Plotly Figure object. In case a single country is currently selected or no countries are selected, the chart is created using the px.bar interface, while in case two or more countries are selected, the Figure is created by adding one **trace** for each

country. Each of the traces are created using the go.bar interface. The Figure is stored in the global variable **g_last_fig** to be used in the next steps for the download operations.

In this step of the dashboard the tabs that are present in the title bar are managed. In particular, the **on_click_tab** function is now modified to manage the display of the 'Chart' or 'Table' view of the filtered dataset:

```
# Click on a tab of the title: change the current view
def on_click_tab(arg):
    global g_view
    if arg == 'Chart':
        g_view = 0
    elif arg == 'Table':
        g_view = 1
    else:
        g_app.snackbar(arg)
    displayCurrentView()
```

This function sets the global variable **g_view** to the correct value (0=Chart, 1=Table, 2=Static Map, 3=Dynamic Map) and calls the **displayCurrentView** function that is responsible for deciding which display function to call:

```
# Display the current view in the outDisplay
def displayCurrentView():
    if g_view == 0:
        displayChart()
    elif g_view == 1:
        displayDatatable()
    elif g_view == 2:
        pass
    else:
        pass
```

At this stage the dashboard looks like:

2.1.1.5 Step 5: Add the SVG Static Map View

This step adds the SVG Static Map View to the dashboard. This type of map represents the countries of Europe as a drawing, without any possibility of doing pan or zoom operations.

In the first cell the list of modules imported from the vois library is updated:

```
from vois.vuetify import app, selectMultiple, label, datatable, toggle,
                                 tooltip, slider, switch
from vois import svgMap
```

Two new Output widgets are created. They will partition the **outDisplay** Output in a left part containing controls to customize the Map views, and a right part to display the Map view itself:

```
outMapControls = widgets.Output(layout=Layout(width=widthControls, min_
↪width=widthControls,
                                              height=height_net, border=border))
outMap         = widgets.Output(layout=Layout(width='90%', height=height_net,␣
↪border=border))
```

Then the controls are created:

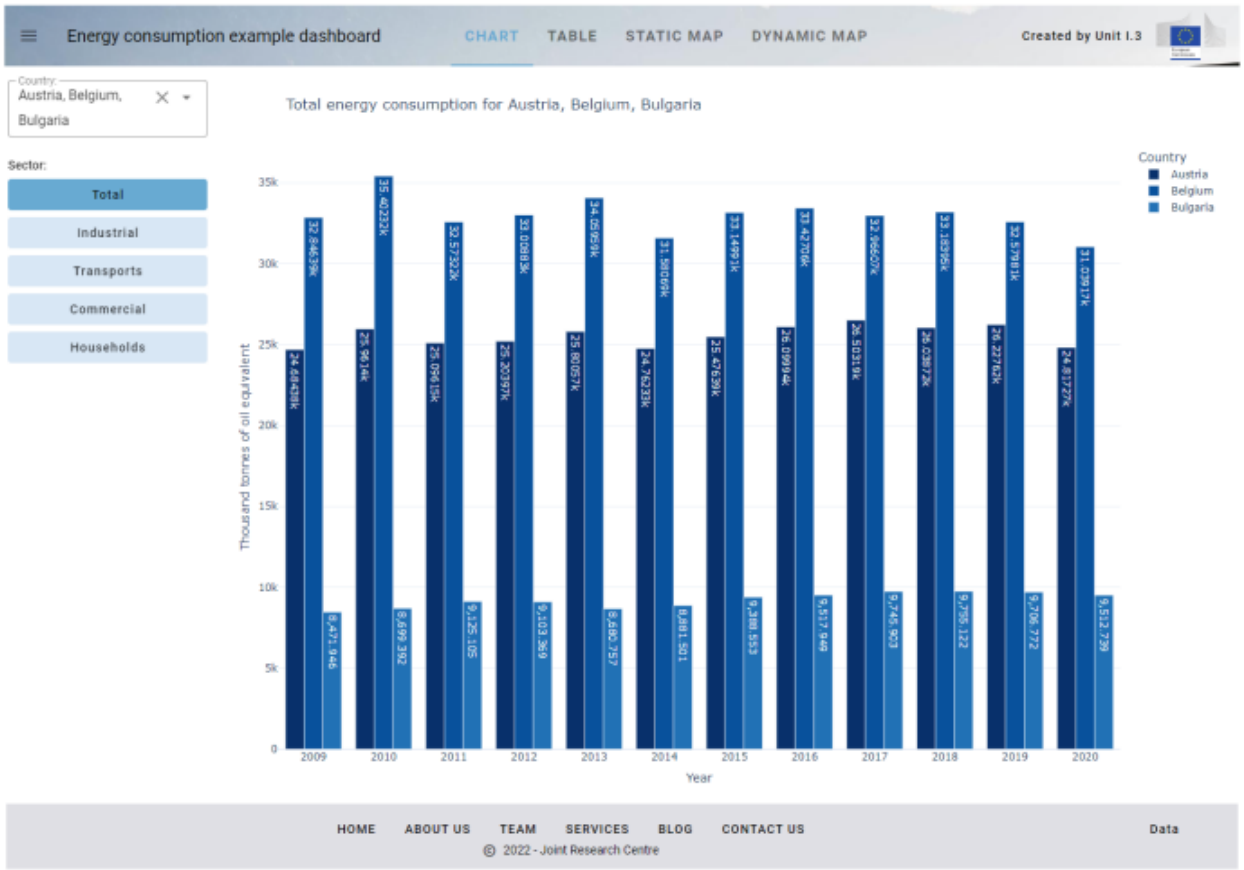


Fig. 2.9: Appearance of the dashboard on step 4

```
labelYear  = label.label('Select the Year:', textweight=400, height=26, margins=3,␣
↪margintop=0)
sliderYear = slider.slider(g_year, g_minyear,g_maxyear, onchange=onchange_year)

popswitch = switch.switch(g_usepop, "Normalize by Population", onchange=on_popswitch_
↪change)
```

The *slider.slider* instance will enable the user to select the year to be displayed on the map, and the *switch.switch* instance is meant to enable the display of data normalized by the population of the respective country (i.e. energy consumption per 100K inhabitants).

When the user changes one of these settings, these two functions are called to update the Map view:

```
# Selection of a year
def onchange_year(value):
    global g_year
    g_year = value
    UpdateDataframe()
    displayCurrentView()

# Switch among total values or values normalized by population
def on_popswitch_change(arg):
    global g_usepop
    g_usepop = not g_usepop
    UpdateDataframe()
    displayCurrentView()
```

The functions **displayCurrentView** and **on_click_tab** are updated to manage the selection of the 'Static Map' tab in the title bar:

```
def displayCurrentView():
    if g_view == 0:
        displayChart()
    elif g_view == 1:
        displayDatatable()
    elif g_view == 2:
        displayStaticMap()
    else:
        pass

...

g_maintabs = ['Chart', 'Table', 'Static Map', 'Dynamic Map']

# Click on a tab of the title: change the current view
def on_click_tab(arg):
    global g_view
    if arg == g_maintabs[0]:
        g_view = 0
    elif arg == g_maintabs[1]:
        g_view = 1
    elif arg == g_maintabs[2]:
        displayMapControls()
        g_view = 2
```

(continues on next page)

(continued from previous page)

```
    else:
        displayMapControls()
        g_view = 3
    displayCurrentView()
```

A new global variable named **g_maintabs** is added to store the names of the 4 tabs in the Title Bar so that those strings are written in a single place and their management becomes easier in case of changes.

The **displayStaticMap** function is a new function to update the Map Output with a static map of Europe. It is created by deriving a **dfmap** Pandas Dataframe where the 'value' column is the total consumption or the consumption per 100K inhabitants, depending on the curren status of the population switch. The map is calculated using the *svgMap.svgMapEurope()* that returns a string in SVG (see SVG: Scalable Vector Graphics). This string is displayed using call to display(HTML()):

```
def displayStaticMap():
    global g_last_svg

    # Prepare the pandas dataframe
    dfmap = g_df[(g_df['TIME_PERIOD']==int(g_year))&(g_df['nrg_bal']==g_sector)].copy()
    dfmap = dfmap[dfmap['geo'].isin(svgMap.country_codes)]
    if g_usepop:
        dfmap['value'] = 100000.0 * dfmap['OBS_VALUE'] / dfmap['Population2021']
        legendunits = 'KTOE per 100K inhabit.'
    else:
        dfmap['value'] = dfmap['OBS_VALUE']
        legendunits = g_units

    colorlist = g_colorsequence[::-1]   # From lighter to darkest!

    # Display the map
    outMap.clear_output(wait=True)
    with outMap:
        selected = []
        if g_countries == ['EU27_2020']: selected = []
        else:                            selected = g_countries

        g_last_svg = svgMap.svgMapEurope(dfmap, code_column='geo', value_column='value',␣
↪codes_selected=selected, stroke_selected='red',
                                         colorlist=colorlist, stdevnumber=2.0,
                                         onhoverfill='#f8bd1a', width=1480-2*widthinpx,␣
↪stroke_width=3.0, hoveronempty=False,
                                         legendtitle=str(g_year) + ' ' + g_sectorName[g_
↪sector], legendunits=legendunits)
        display(HTML(g_last_svg))
```

In the call to *svgMap.svgMapEurope()* function, the column named **geo** of the **dfmap** DataFrame is used to match the corresponding polygon using the EUROSTAT Country Codes, and the related numerical value taken from the **value** column is assigned to the polygon. With all the polygons having their values assigned, the mean and standard deviation of all the values are calculated, and the range of values [mean - 2*stddev, mean + 2*stddev] is linearly assigned to the input colorlist (the Plotly px.colors.sequential.Blues).

The codes of the countries selected in the selectMultiple object are passed to the svgMapEurope function to highlight in red the border of the countries on the map drawing and on the legend.

This is the apperance of the dashboard after the execution of the Step 5 notebook cells:

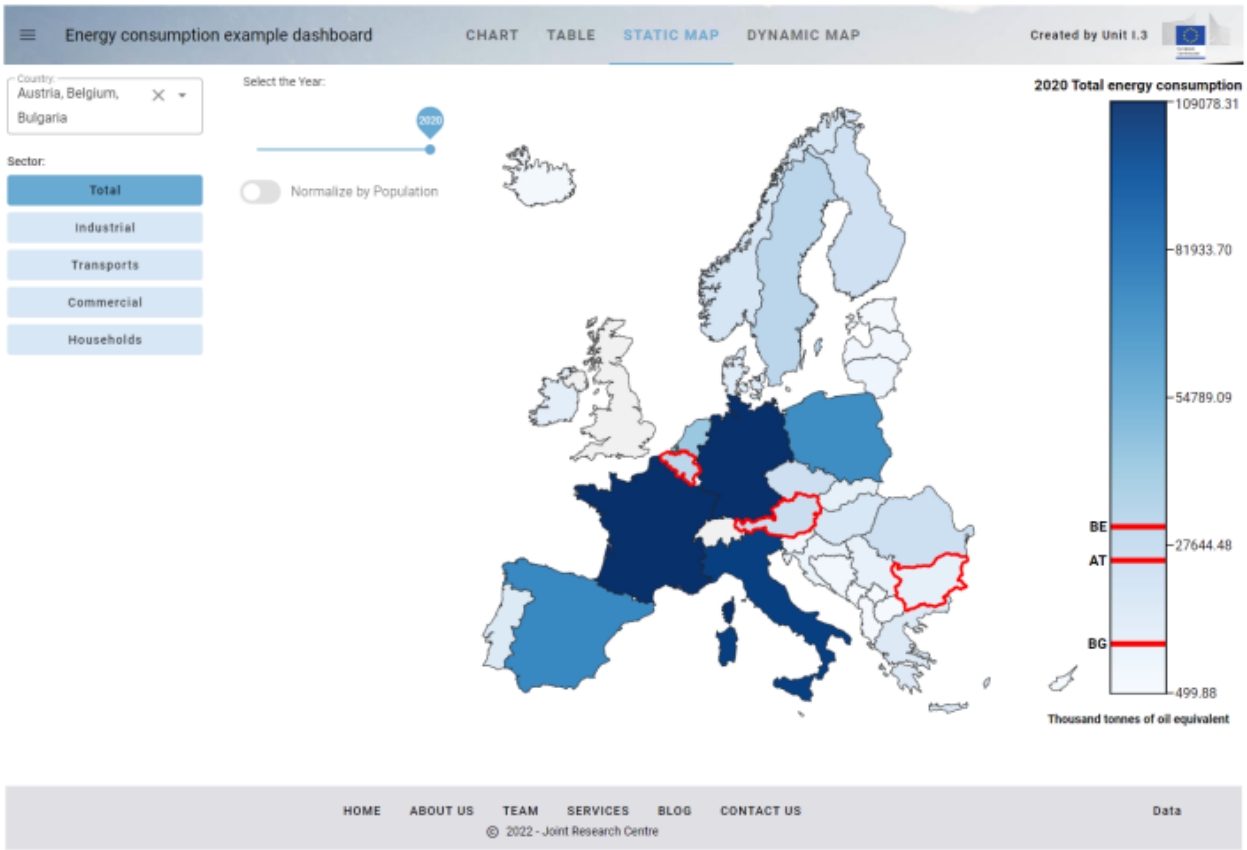


Fig. 2.10: Appearance of the dashboard on step 5

2.1.1.6 Step 6: Add the Dynamic Map View

This step adds the Dynamic Map View to the dashboard. The Map is created using the ipyleaflet widget library which is a porting into the Jupyter world of the widely used Leaflet JS library. Maps created in this way allow the users to do panning and zooming operation and to click on the features displayed in the map to get information. Vois library has a `leafletMap` module that exports some functions for adding custom vector dataset to a ipyleaflet interactive map.

In the first cell of the notebook, some additional import commands are added:

```
from ipyleaflet import basemaps
from vois import svgMap, leafletMap, svgUtils, geojsonUtils
```

The basemaps module allows for the selection of a backdrop dataset to be displayed in a ipyleaflet map, while the modules `leafletMap` and `geojsonUtils` are used for importing a geospatial dataset in geojson format inside the map and `svgUtils` is used to display a graduated legend that helps in understanding the colors assigned to the country polygons.

The same Pandas DataFrame calculated in the previous step as input to the Static Map is used to define the colors to be assigned to the polygons of the Dynamic Map. For this reason some lines of code that in the previous step were written inside the **displayStaticMap** function, are now extrapolated and inserted in a new function that returns the DataFrame used by both the Static and the Dynamic Map:

```
# Prepare the pandas dataframe for the Map display (returns a df)
def dataframeForMap():
    dfmap = g_df[(g_df['TIME_PERIOD']==int(g_year))&(g_df['nrg_bal']==g_sector)].copy()
    dfmap = dfmap[dfmap['geo'].isin(svgMap.country_codes)]
    if g_usepop:
        dfmap['value'] = 100000.0 * dfmap['OBS_VALUE'] / dfmap['Population2021']
    else:
        dfmap['value'] = dfmap['OBS_VALUE']
    return dfmap
```

The important addition of this step of the dashboard is the **displayDynamicMap** function. Its main content is the creation of the Dynamic Map starting from the dfmap returned by the **dataframeForMap** function call and a geojson file named **ne_50m_admin_0_countries.geojson** that is stored under the **data** folder:

```
g_map = leafletMap.geojsonMap(dfmap,
                              './data/ne_50m_admin_0_countries.geojson',
                              'ISO_A2_EH',
                              code_column='geo',
                              value_column='value',
                              codes_selected=selected,
                              stroke_selected='red',
                              colorlist=colorlist,
                              stdevnumber=2.0,
                              stroke_width=0.6,
                              stroke='#010101',
                              width='70%',
                              height=height,
                              center=g_center,
                              zoom=g_zoom,
                              basemap=basemaps.Esri.WorldTopoMap,
                              style      ={'opacity': 1, 'dashArray': '0', 'fillOpacity
↪': 0.85},
                              hover_style={'opacity': 1, 'dashArray': '0', 'fillOpacity
↪': 0.99})
```

The **ne_50m_admin_0_countries.geojson** file was obtained from the Natural Earth web site and contains polygons for all the countries of the world. Unfortunately, this dataset uses a different coding for the countries: instead of the EUROSTAT Country Codes, it uses the ISO 3166 standard in the attribute named 'ISO_A2_EH'. The only practical consequence of having two sets of codes is that we have to cope with the two different codes that are assigned to Greece, which is coded **EL** in the EUROSTAT Country Code and **GR** in the ISO 3166 standard. This is managed by these two lines:

```
dfmap['geo'].replace('EL','GR',inplace=True)

selected = ['GR' if x=='EL' else x for x in selected]
```

These two lines substitute all occurrencies of 'EL' with 'GR' before calling the **leafletMap.geojsonMap** function.

These lines are added to observe any zoom and panning event in the dynamic map:

```
g_map.observe(map_on_bounds_changed, names='bounds')
```

The **map_on_bounds_changed** is called any time that the map bounds (the east/west/south/north coordinates) are changed by a user action. Its code stores the center and the zoom level into two global variables **g_center** and **g_zoom**:

```
# Store center and zoom at each zoom or panning of the user
def map_on_bounds_changed(args):
    global g_center, g_zoom
    if not g_map is None:
        g_center = g_map.center
        g_zoom   = g_map.zoom
```

These values are then used any time that the Dynamic Map is updated (for instance when the user selects a different year) to recreate the new map with the same visible portion of the globe on the screen.

In the call to *leafletMap.geojsonMap()* function, the column named **geo** of the **dfmap** DataFrame is used to match the corresponding polygon using the ISO 3166 standard, and the related numerical value taken from the **value** column is assigned to the polygon. With all the polygons having their values assigned, the mean and standard deviation of all the values are calculated, and the range of values [mean - 2*stddev, mean + 2*stddev] is linearly assigned to the input colorlist (the Plotly px.colors.sequential.Blues). A call to the *svgUtils.graduatedLegend()* is then used to create a graduated colors legend in SVG format to be displayed on the right side of the map:

```
svg = svgUtils.graduatedLegend(dfmap, code_column='geo', value_column='value',
                               codes_selected=selected, stroke_selected='red',
                               colorlist=colorlist, stdevnumber=2.0,
                               legendtitle=str(g_year) + ' ' + g_sectorName[g_sector],
                               legendunits=legendForUnits(),
                               fontsize=15, width=340, height=heightinpx-60)
```

This is the part of the new code that clears the **outMap** Output widget and then displays in it both the ipyleaflet Map and the SVG graduated color legend:

```
# Display the legend
outlegend = widgets.Output(layout=Layout(width='360px',height=height))
with outlegend:
    display(HTML(svg))

# Display the map
outMap.clear_output(wait=True)
with outMap:
    display(widgets.HBox([g_map,outlegend]))
```

The functions **displayCurrentView** and **on_click_tab** are updated to manage the selection of the 'Dynamic Map' tab in the title bar:

```
def displayCurrentView():
    if g_view == 0:
        displayChart()
    elif g_view == 1:
        displayDatatable()
    elif g_view == 2:
        displayStaticMap()
    elif g_view == 3:
        displayDynamicMap()
...

g_maintabs = ['Chart', 'Table', 'Static Map', 'Dynamic Map']

# Click on a tab of the title: change the current view
```

(continues on next page)

(continued from previous page)

```
def on_click_tab(arg):
    global g_view, g_center, g_zoom
    if arg == g_maintabs[0]:
        g_view = 0
    elif arg == g_maintabs[1]:
        g_view = 1
    elif arg == g_maintabs[2]:
        displayMapControls()
        g_view = 2
    else:
        g_center = [56,11]
        g_zoom   = 4
        displayMapControls()
        g_view = 3
    displayCurrentView()
```

This is the apperance of the dashboard after the execution of the Step 6 notebook cells:

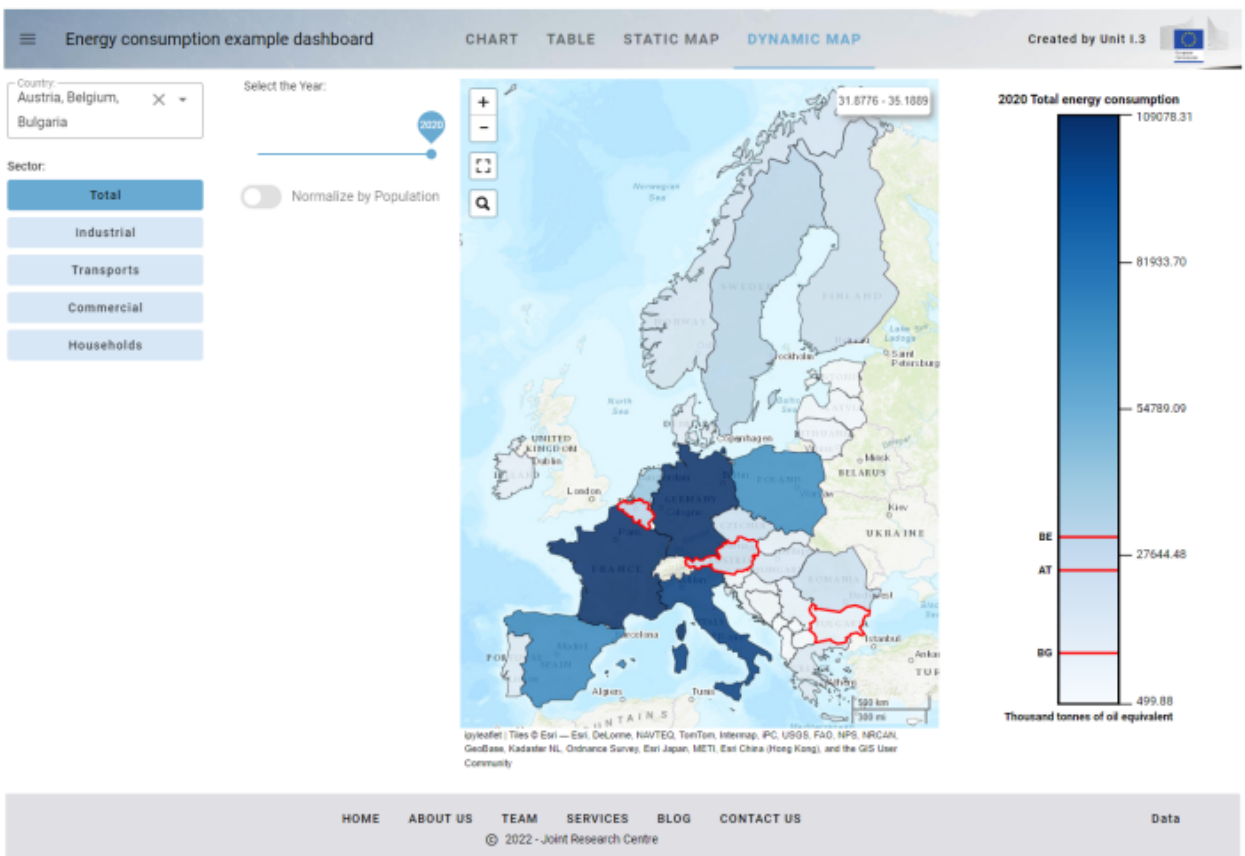


Fig. 2.11: Appearance of the dashboard on step 6

2.1.1.7 Step 7: Add the functions for downloading chart, table and map

The 7th step adds the functions for downloading chart, table and map view.

The first cell of the notebook has some new import instructions:

```
from datetime import datetime
import io
from cairosvg import svg2png
```

These new lines import modules used by the export code (svg2png is a module of the cairosvg package that converts SVG drawings to raster PNG format). From the vois library, the module **fab** is imported:

```
from vois.vuetify import app, selectMultiple, label, datatable,
                                 toggle, tooltip, slider, switch, fab
```

fab is a module of the vois library that helps in creating Floating-Action-Button, i.e. buttons that are positioned in absolute mode on the page and that display a menu when hovered.

In the last cell of the notebook, just after the creation of the **g_app** instance, the download fab button is created:

```
b = g_app.fab(left='96%', top='108px',
              items=['Download Chart', 'Download Table', 'Download Map'],
              onclick=[ondownloadCHART,ondownloadCSV,ondownloadMAP])
```

This button is positioned in the topright portion of the screen and, when hovered, it shows three menu items for downloading the chart, the table and the map view.

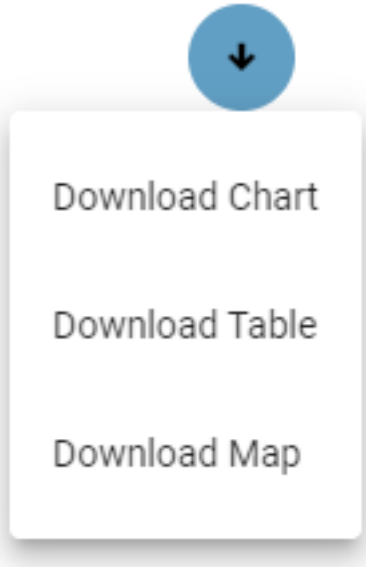


Fig. 2.12: Download functions in the fab button

The corresponding download functions are declared in a new cell that contains these lines for the download of the Poltly chart:

```
# Download Chart in PNG
def ondownloadCHART():
    global g_view
```

(continues on next page)

(continued from previous page)

```
    if g_view != 0:
        g_view = 0
        g_app.setActiveTab(g_view)
        on_click_tab(g_maintabs[g_view])

    if not g_last_fig is None:
        filename = 'Energy_' + datetime.today().strftime('%Y-%m-%d_%H-%M-%S')
        png = g_last_fig.to_image(format="png", width=1800)
        buf = io.BytesIO(png)
        buf.seek(0)
        barray = buf.read()
        #with open("chart.png", "wb") as file:
        #    file.write(barray)
        g_app.downloadBytes(barray,'%s.png' % filename)
```

These are the lines for the download of the tabular data in CSV format:

```
# Download data in CSV format
def ondownloadCSV():
    if not g_dtfiltered is None:
        filename = 'Energy_' + datetime.today().strftime('%Y-%m-%d_%H-%M-%S')
        df = g_dtfiltered.copy()
        df = df.reset_index(drop=True)
        buf = io.StringIO()
        df.to_csv(buf)
        buf.seek(0)
        text = buf.getvalue()
        g_app.downloadText(text,"%s.csv" % filename)
```

These are the lines to download the static map in PNG format:

```
# Download Map in PNG
def ondownloadMAP():
    global g_view

    g_app.dialogWaitOpen(text='Please wait for Map export...')

    if g_view != 2:
        g_view = 2
        g_app.setActiveTab(g_view)
        on_click_tab(g_maintabs[g_view])

    if not g_last_svg is None:
        filename = 'Energy_' + datetime.today().strftime('%Y-%m-%d_%H-%M-%S')
        svg_picture = g_last_svg
        svg_picture = svg_picture.replace(' ','&amp;nbsp;')
        svg_picture = svg_picture.replace('style="font-family: Roboto;"','')
        png = svg2png(bytestring=svg_picture, parent_width=1850, parent_height=600)
        buf = io.BytesIO(png)
        buf.seek(0)
        barray = buf.read()
```

(continues on next page)

(continued from previous page)

```
        #with open("map.png", "wb") as file:
        #    file.write(barray)
        g_app.downloadBytes(barray,'%s.png' % filename)

    g_app.dialogWaitClose()
```

All the three download functions use the g_app methods *app.app.downloadText()* and *app.app.downloadBytes()* to download textual and binary files to the user local machine. The download of the Static Map, given that the download operations can take some seconds, is enclosed in the opening and closing of a *dialogWait.dialogWait* invoked through the **g_app** instance:

```
g_app.dialogWaitOpen(text='Please wait for Map export...')

...

g_app.dialogWaitClose()
```

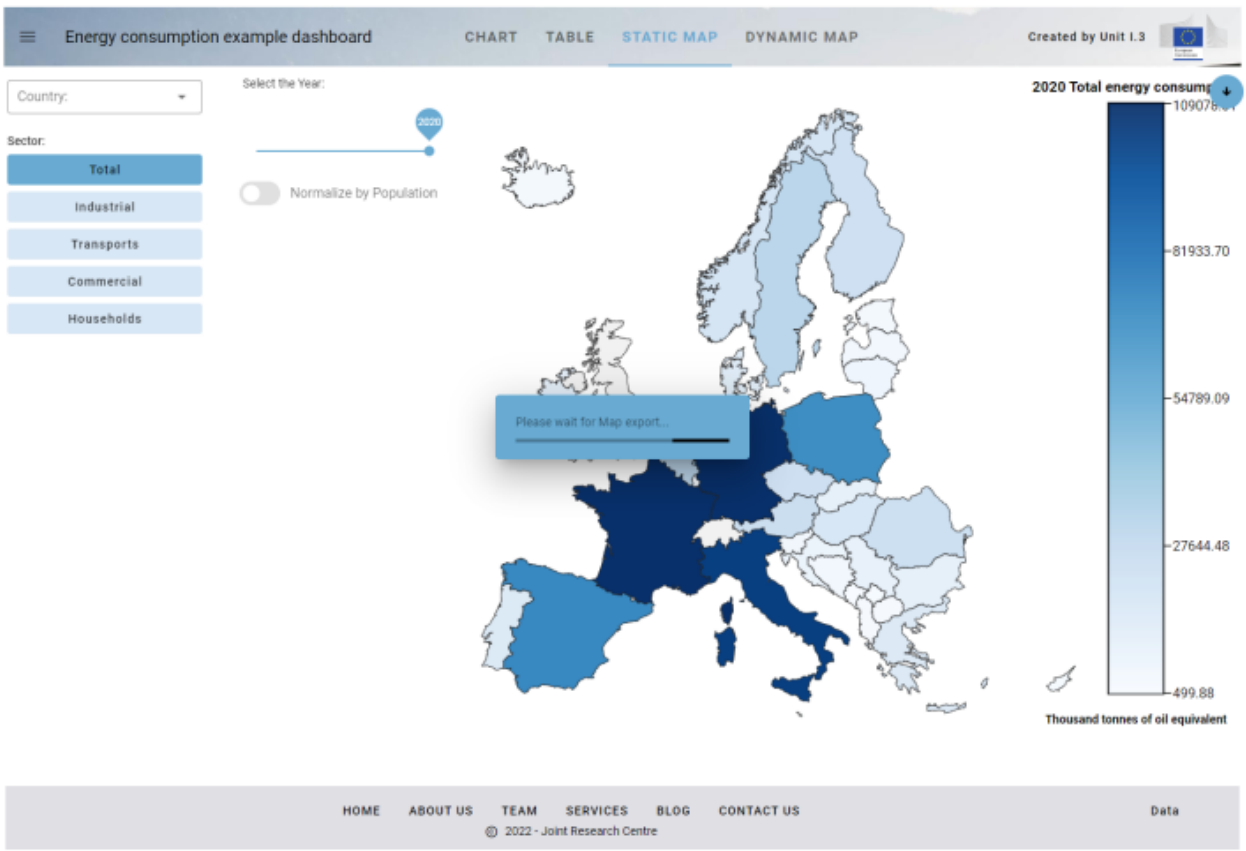


Fig. 2.13: Dialog-box displayed during the preparation of the data for the download of the Map

2.1.1.8 Step 8: Manage the parameters passed in the URL and open external URLs

This step adds the management of the parameters passsed in the URL and adds the opening of an external URLs when the user clicks on the EC logo in the title bar.

In developing real dashboards, there can be the need to manage the parameters passed in the URL that launched the application. It can be useful, for instance, to have a dashboard that provides different views of the same data, that are invoked by passing a parameter in the URL (for instance adding "?view=1" or "?view=2" to the URL). The python code of the dashboard can read these parameters and behave accordingly.

In order to add this new functionality, only the last cell of the notebook is modified in this 8th step. These lines of code are added just after the creation of the g_app and the fab button:

```
# Read the URL parameters
g_view   = int( g_app.urlParameter('view',   g_view))
g_sector = str( g_app.urlParameter('sector', g_sector))
g_year   = int( g_app.urlParameter('year',   g_year))
g_usepop = bool(g_app.urlParameter('usepop', g_usepop))
countries = str( g_app.urlParameter('countries', ''))
if len(countries) <= 0: g_countries = []
else:                   g_countries = countries.split(',')
```

The function *app.app.urlParameter()* is called five times to read one URL parameter given its name. The values read from the URL are assigned to the global variables **g_view**, **g_sector**, **g_year**, **g_usepop** and **g_countries**. These global variables are used to set the active tab in the title bar of the g_app and to change the current value of the input widgets:

```
g_app.setActiveTab(g_view)
on_click_tab(g_maintabs[g_view])
if g_sector in g_sectors:
    selsector.value = g_sectors.index(g_sector)
sliderYear.value = g_year
popswitch.value  = g_usepop
selcountry.value = g_countries
```

The call to:

```
UpdateDataframe()
displayCurrentView()
```

updates the content displayed in the Output widgets according to the parameters passed in the URL.

To fully manage the URL parameters, it is also useful to update the URL in the browser address bar when the user operates on the widgets of the dashboard (it is a function similar to what is implemented by web sites like https://www.google.com/maps/: panning and zooming on the map updates the address bar of the browser by adding the center position and the zoom level of the map). This can be obtained by calling the helper function *urlUpdate()* to pass the part of the URL to modify after the '?', as in the **urlUpdate** function:

```
# Update the URL
def urlUpdate():
    url = "?view=%d&countries=%s&sector=%s&year=%d&usepop=%d" % (int(g_view),
                                                                 ",".join(g_countries),
                                                                 str(g_sector),
                                                                 int(g_year),
                                                                 int(g_usepop))
    g_app.urlUpdate(url)
```

The **urlUpdate** function is then called in all the functions that are executed when the user operates a selection on the widgets control, like in the functions **onchange_country**, **onchange_sector**, **onchange_year**, **on_popswitch_change** and **on_click_tab**. In this way the URL displayed while the dashboard is active, always reflects the current status and can be, for instance copy/pasted to return to the same situation at a later stage or pass the complete URL to a colleague.

Finally, in the python function that manages the click on the logo, the call to open an external page is added, instead of simply send a snackbar message to the user, by using the *app.app.urlOpen()* method of the *app.app* class:

```
# Click on the logo
def on_click_logo():
    g_app.urlOpen('https://ec.europa.eu/info/index_en')
```

2.1.1.9 Step 9: Add an interactive AnimatedPieChart in SVG to select the consumption sector

The 9th step of the Energy Consumption dashboard adds an interactive AnimatedPieChart in SVG to select the consumption sector. The contribution of this addition is not important for the new functionality that it provides to the user, but as an example of an **interactive** SVG drawing inside the dashboard. The SVG added is 'interactive' in the sense that it is not only a drawing with an animation, but it manages user actions like click events, to provide some functionality to the application. This function is based on small but very important component called ipyevents, that is able to capture events occurring in the user client browser (like the user clicking on a slice of the pie), and pass it back to the python kernel to execute a function when it happens. This allows dashboard developers to complement their applications with reactive custom drawing that can fill the gaps of the standard charting libraries like Plotly or Bokeh: when a chart that is requested is not available in those libraries, an interactive SVG drawing can be added to the dashboard.

Here is a picture of the animated pie chart that is added in this step.

The chart graphically represents the subdivision of the energy consumptions for the current countries selection in the different sectors (transport, commercial, etc.). By clicking in one of the pie slices, the corresponding sector is selected (similarly to the clicking on the buttons of the sectors toggle widget)

Let's examine the code added to the notebook to obtain the new function. First of all, a new Output widget named outAnimation is created in the second cell of the notebbook and the Output widgets dimensioning for the Map and the Map controls are slightly modified:

```
widthmapinpx     = 360
widthMapControls = '%dpx' % widthmapinpx
outMapControls = widgets.Output(layout=Layout(width=widthMapControls, min_
↪width=widthMapControls, height=height_net, border=border))
outMap         = widgets.Output(layout=Layout(width='90%', height=height_net,␣
↪border=border))

widthanimpx  = widthmapinpx - 20
widthanim    = '%dpx' % (widthanimpx+10)
outAnimation = widgets.Output(layout=Layout(width=widthanim, min_width=widthanim,␣
↪height=widthanim, border=border))
```

Then, a new cell is added containing the code to create and display the animated pie chart:

```
# Display Pie Chart animation
def displayAnimation():

    outAnimation.clear_output(wait=True)
```

(continues on next page)

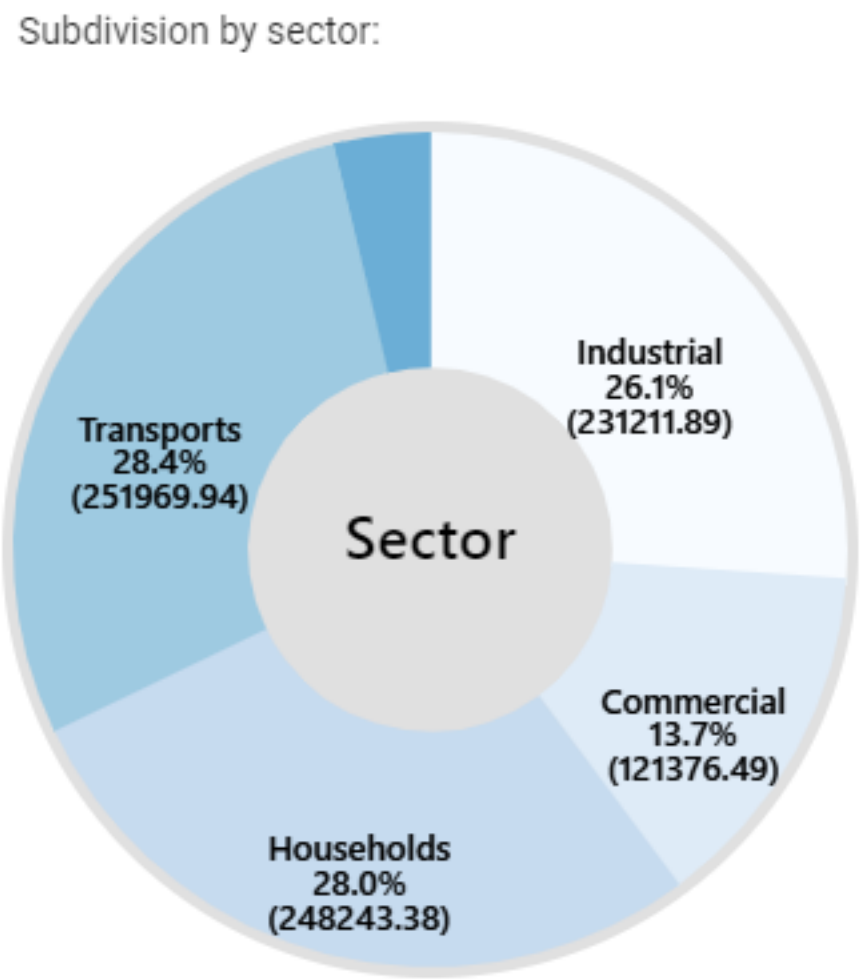


Fig. 2.14: The animated pie chart added to the dashboard

(continued from previous page)

```
    if len(g_countries) == 0: codes = ['EU27_2020']
    else:                     codes = g_countries
    df_country_year = g_df[(g_df['geo'].isin(codes))&(g_df['TIME_PERIOD']==g_year)]
    df_country_year = df_country_year.groupby(["nrg_bal"])["OBS_VALUE"].sum().to_frame().
↪reset_index()
    sectors = list(df_country_year['nrg_bal'])
    values  = list(df_country_year['OBS_VALUE'])

    chartvalues  = []
    chartlabels  = []
    chartsectors = []
    if 'FC_E' in sectors:
        totalindex = sectors.index('FC_E')
        totalvalue = values[totalindex]

        total = 0.0
        for s,v in zip(sectors,values):
            if s != 'FC_E':
                total += v
                chartvalues.append(round(v,2))
                chartlabels.append(g_sectorTitle[s])
                chartsectors.append(s)

        chartvalues.append(round(totalvalue - total,2))
        chartlabels.append('Other')
        chartsectors.append(None)


    def onclick(arg):
        newsector = chartsectors[arg]
        if not newsector is None:
            g_sector = newsector
            selsector.value = g_sectors.index(g_sector)

    out, txt = svgUtils.AnimatedPieChart(values=chartvalues, labels=chartlabels,␣
↪decimals=1,
                                         centerfontsize=28, fontsize=16, textweight=500,␣
↪colors=px.colors.sequential.Blues, backcolor='#e0e0e0',
                                         centertext='Sector', onclick=onclick,␣
↪dimension=widthanimpx-15, duration=1.0)

    with outAnimation:
        display(out)
```

The first lines of the **displayAnimation** new function extract the information on sector consumptions from the current Pandas DataFrame **g_df**, to create a list of sector names (**chartlabels**) and a corresponding list of numerical values (**chartvalues**). Then the *svgUtils.AnimatedPieChart()* method is called passing these two lists as input data for the creation of the pie chart. A local pyton function is also passed as the **onclick** parameter. This function will be called (thanks to the internal code of the *svgUtils.AnimatedPieChart()* method that uses **ipyevents** to capture the events) at each click on the slices of the pie chart. The onclick function will receive as parameter the label of the clicked slice and this allows the function to update the value of the **selsector** toggle widget.

Note: Please note that, given that the *svgUtils.AnimatedPieChart()* method is meant to capture users events, it returns an Output widget instance. This Output widget is created internally and is passed to the ipyevents events manager to capture user clicks. For those interested in understanding how this works, pleas refer to the source code of the *svgUtils.AnimatedPieChart()* method, and in particular these lines:

```
d = Event(source=out, watched_events=['click'])

def handle_event(event):
    ...

d.on_dom_event(handle_event)
```

2.1.1.10 Step 10: Add minipanel to footer bar and the function to generate and download a report in docx format

The final step adds a minipanel to the footer bar and the function to generate and download a report in Microsoft Word **docx** format containing text, tables and images.

The minipanel is a small widget containing icons that can be added to the footer bar of a *app.app* class instance. These are the lines of code added in the last cell of the notebook to add the minipanel in the creation of the global **g_app** instance:

```
minipanelicons=['mdi-chart-bar',
                'mdi-table-large',
                'mdi-map-legend',
                'mdi-file-chart-outline'],
minipaneltooltips=['Download Chart',
                   'Download Table',
                   'Download Map',
                   'Generate Report in Word .docx format'],
minipanellarge=True, minipanelopen=False,
onclickminipanel=on_click_minipanel,
```

Fig. 2.15: The minipanel added to the footer bar of the application

The minipanel contains four icons, each icon has a tooltip text, and when clicking on one of the icons, the **on_click_minipanel** new function is called, which calls the correct download function, given the index of the clicked icon that it receives as parameter:

```
# Click on the footer minipanel
def on_click_minipanel(arg):
    if   arg == 0: ondownloadCHART()
    elif arg == 1: ondownloadCSV()
    elif arg == 2: ondownloadMAP()
    else:          ondownloadREPORT()
```

Note: All the icons from https://materialdesignicons.com/ site can be used, just by prepending 'mdi-' to their name.

All the free icons from https://fontawesome.com/ site can be used, just by prepending 'fa-' to their name.

To create and export a report in Microwoft Word docx format, these new imports are added in the first cell of the notebook, to load modules from the python-docx package:

```
from docx import Document
from docx.shared import Inches
```

This is the new function **ondownloadREPORT** that is called from the minipanel:

```
def ondownloadREPORT():
    global g_view

    if g_view != 0:
        g_view = 0
        g_app.setActiveTab(g_view)
        on_click_tab(g_maintabs[g_view])

    g_app.dialogWaitOpen(text='Please wait for Word .docx report generation...')

    document = Document()
    document.add_heading('Report from the Energy consumption dashboard', 0)

    # Add a table
    document.add_heading('Filtered table:', level=2)
    df = g_dtfiltered[['nrg_bal', 'siec', 'geo', 'Year', 'Thousand tonnes of oil␣
↪equivalent', 'Country']]
    add_table(df, document)
    document.add_page_break()

    # Add chart as an image
    png = g_last_fig.to_image(format="png", width=1800)
    imagetitle = str(g_year) + ' ' + g_sectorName[g_sector] + ' in ' + legendForUnits()␣
↪+ ':'
    document.add_heading(imagetitle, level=2)
    file = io.BytesIO(png)
    document.add_picture(file, width=Inches(7.0))

    # Add map as an image
    if g_last_svg is None:
        displayStaticMap()

    if not g_last_svg is None:
```

(continues on next page)

(continued from previous page)

```
        imagetitle = str(g_year) + ' ' + g_sectorName[g_sector] + ' in ' +␣
↪legendForUnits() + ':'
        document.add_heading(imagetitle, level=2)
        add_image(g_last_svg, document)

    buf = io.BufferedRandom(io.BytesIO())
    document.save(buf)
    buf.seek(0)

    barray = buf.read()
    g_app.downloadBytes(barray,'Energy.docx')

    g_app.dialogWaitClose()
```

The function creates an example report containing some text, a table and two images (the chart and the static map). Then the *app.app.downloadBytes()* method is used to transfer the bytes of the report to the user local machine.

2.1.1.11 Step 10 dark: Creation of the "dark" version of the final dashboard

It can be useful to be able to create Voilà dashboard that use the "Dark Theme". To obtain a "dark" dashboard, some additional steps have to be performed. First of all, the Step 10 example notebook EnergyConsumption.Final.ipynb has been copied to another notebook and named **EnergyConsumption.Final.ThemeDark.ipynb**. The first action to convert the dashboard to the "Dark Theme" is to edit the .ipynb file using the text editor and add this settings to the "metadata" part of the notebook JSON:

```
,
 "voila": {
  "theme": "dark"
 }
```

as shown in this screenshot:

Then the first cell of the notebook is modified by adding the global variable **g_dark** with value True and changing **settings.dark_mode** to True. The first and second color are also changed, as well as the background color of the tooltips (see *settings*):

```
g_dark = True

from vois.vuetify import settings
settings.dark_mode         = g_dark
settings.color_first       = '#68aad2'
settings.color_second      = '#1c4056'
settings.button_rounded    = False
settings.tooltip_backcolor = '#424242'
```

Then, immediately after the import of the ipyvuetify package, the dark_mode is selected for the widgets of that library:

```
# Set the dark theme for ipyvuetify!!!
import ipyvuetify as v
v.theme.dark = g_dark
```

The calls to `svgUtils.svgMapEurope()` and to *svgUtils.graduatedLegend()* need to have the dark parameter set to True, so that the border of the countries and the legends are displayed converting 'black' to 'white'. The same is

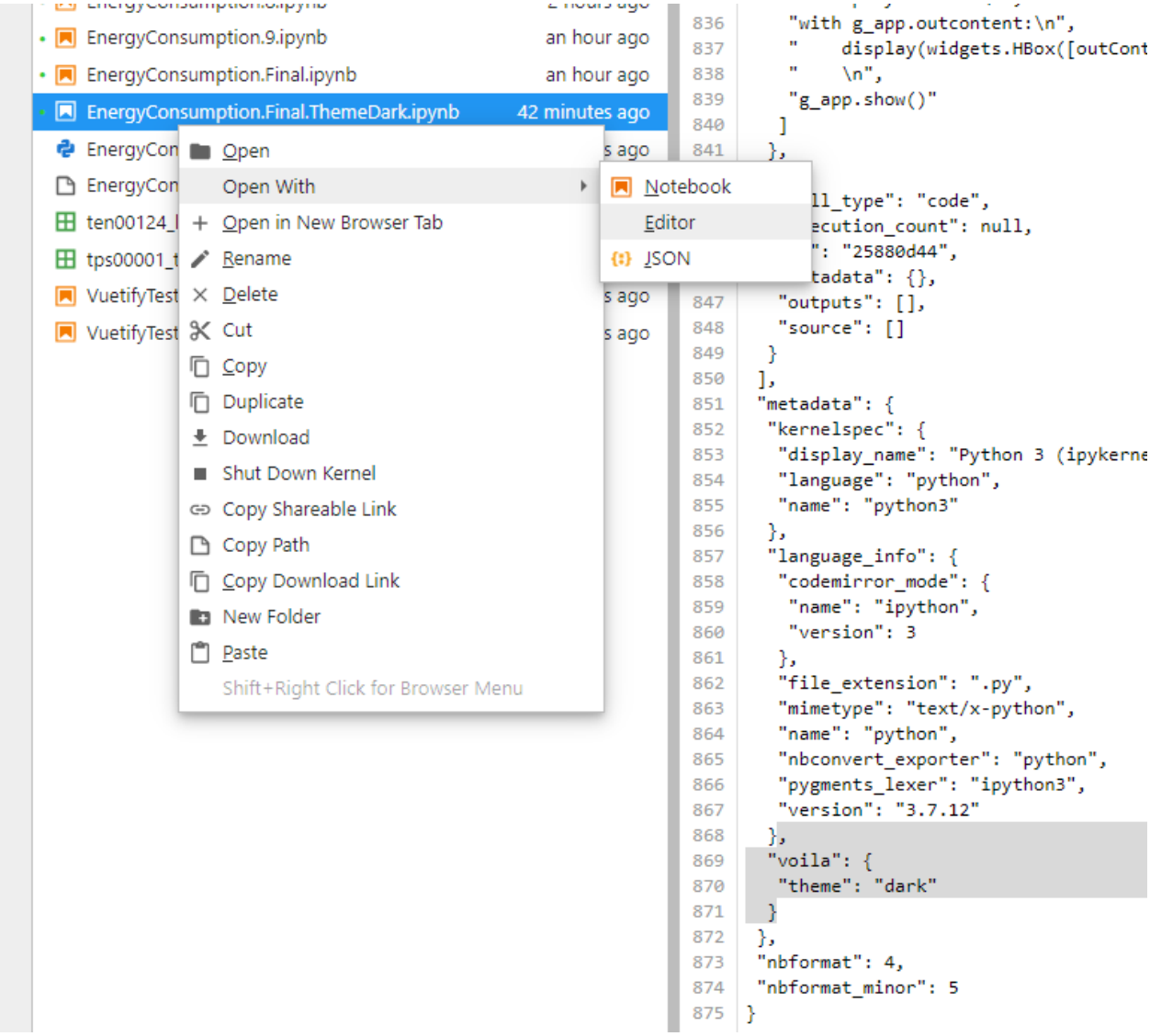


Fig. 2.16: Open in editor the source JSON notebook and add the “voila” “theme” “dark” setting.

done in the call to create the **g_app** instance of the *app.app* class, to draw as white the texts of the title bar and of the footer bar:

```
dark=g_dark,
footerdark=g_dark,
```

The **template='plotly_dark'** parameter is added to the methods called from the Plotly library inside the **displayChart** function to select the dark theme.

Then the backcolor of the call to *svgUtils.AnimatedPieChart()* in the **displayAnimation** function is changed, as well as the basemap for the Dynamic Map that is changed to **basemaps.CartoDB.DarkMatter**.

Here is the final dashboard changed to use the dark theme:

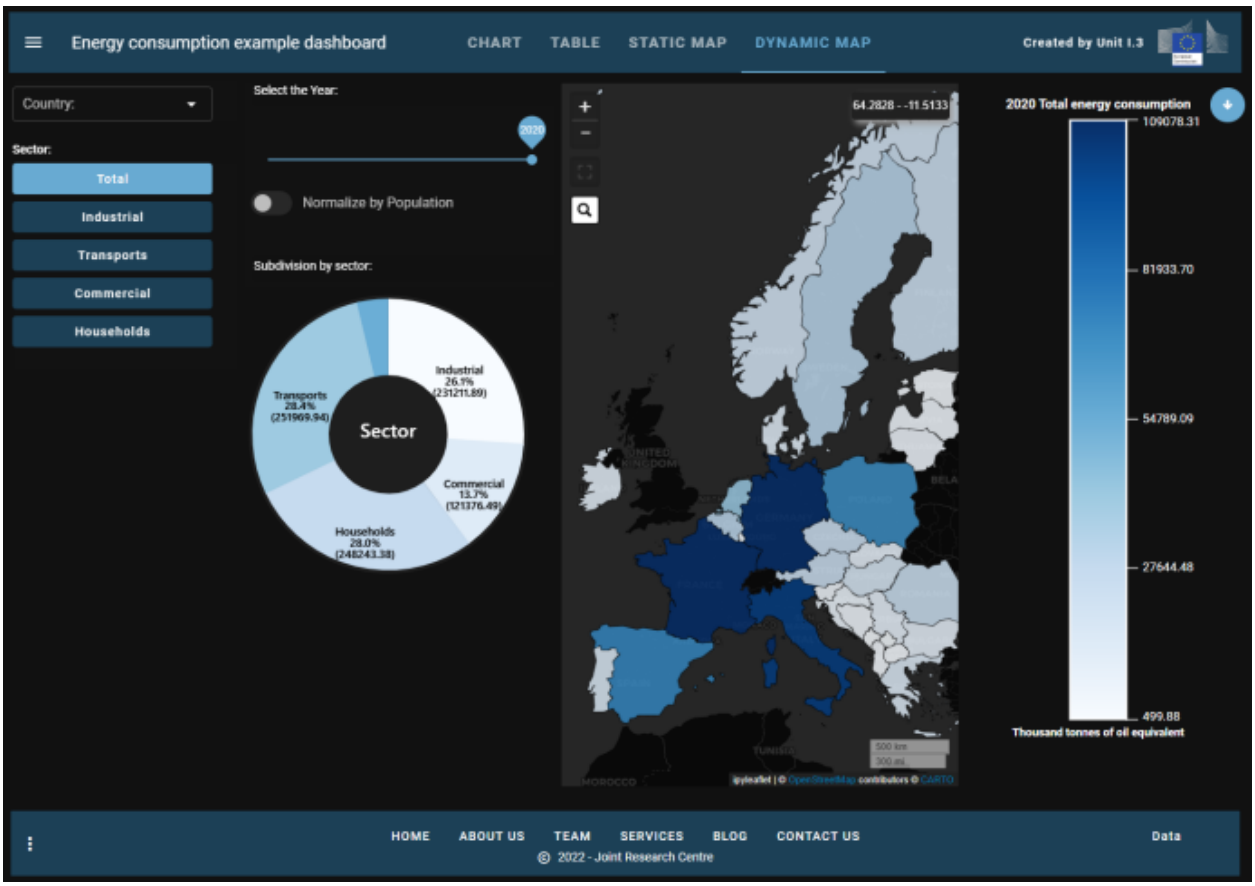


Fig. 2.17: Dark version of the final dashboard

2.2 MultiPage Demo

New classes added to the **vois library** allow for easy creation of multi-pages applications: *mainPage.mainPage* class and *page.page* class.

The mainPage class creates a fullscreen page displaying a background image, a title/subtitle/logo and some buttons. Each button on the mainPage can open a specific page.

A minimal example of a multi-page application can be created by running this example code:

```
from vois.vuetify import mainPage, page
from random import randrange

from ipywidgets import widgets, Layout

output = widgets.Output(layout=Layout(width='0px', height='0px'))
display(output)

title     = 'MultiPage Demo'
logo      = 'https://jeodpp.jrc.ec.europa.eu/services/shared/pngs/BDAP_
↪Logo1024transparent.png'
copyright = 'European Commission - Joint Research Centre'

def randomPicture():
    return 'https://picsum.photos/seed/%d/200/200'%randrange(1000)

def onclick1():
    p = page.page(title, 'Function 1', output, titlecolor='#808dc4', titledark=True,
                  footercolor='#cccccc', logoappurl=logo, copyrighttext=copyright)
    card = p.create()
    card.children = ['Put widgets into the empty space of the page1']
    p.open()

def onclick2():
    p = page.page(title, 'Function 2', output, titlecolor='#808dc4', titledark=True,
                  footercolor='#cccccc', logoappurl=logo, copyrighttext=copyright)
    card = p.create()
    card.children = ['Put widgets into the empty space of the page2']
    p.open()

m = mainPage.mainPage(title='MultiPage Demo',
        subtitle='Showcase how easy is to create a multi-page app',
        credits="vois library development team",
        titlebox_widthpercent=50, titlebox_opacity=0.2, titlebox_border=0,
        vois_show=True, vois_opacity=0.1,
        button_widthpercent=23, button_heightpercent=14,button_elevation=16, button_
↪opacity=0.6,
        background_image=55,
        background_filter='blur(2px) brightness(1.2) contrast(0.7) sepia(0.05)␣
↪saturate(1.2)',
        creditbox_opacity=0,
        text_color='#222222')

m.addButton('Function 1', tooltip='Tooltip text to display on hover function1',
```

(continues on next page)

(continued from previous page)

```
            image=randomPicture(), onclick=onclick1)

m.addButton('Function 2', tooltip='Tooltip text to display on hover function2',
            image=randomPicture(), onclick=onclick2)

for i in range(3,7): m.addButton('Function %d'%i, image=randomPicture())

m.open()
```

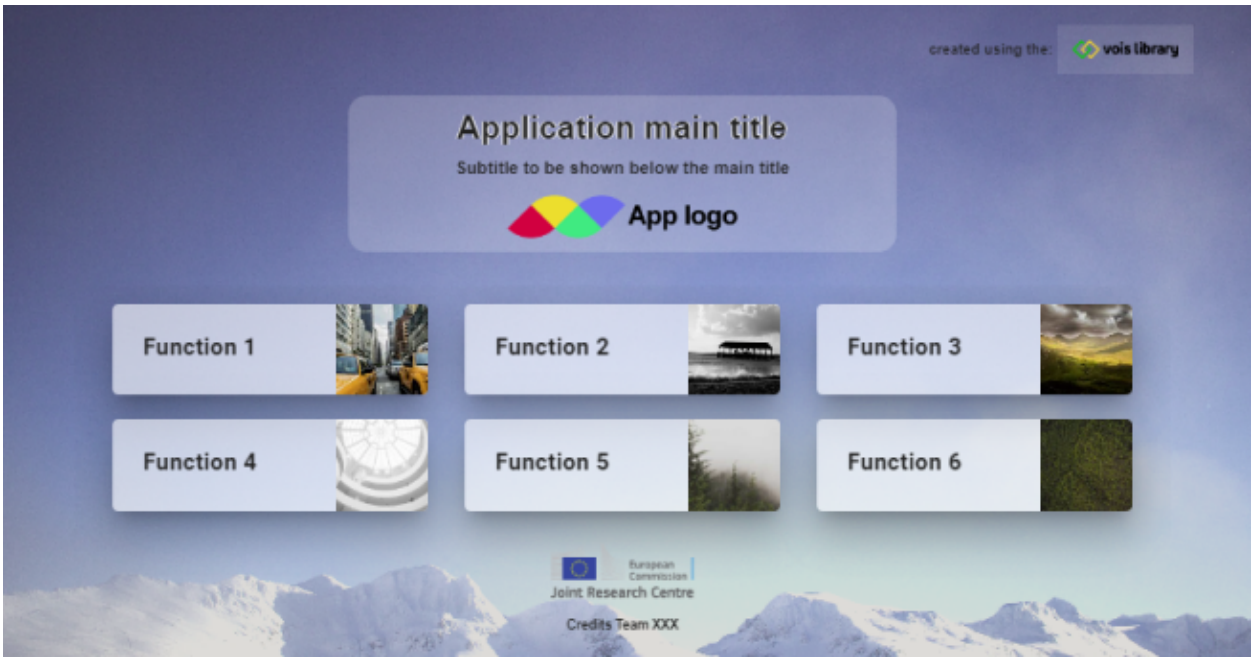


Fig. 2.18: Example of a mainPage

The vois library provides sixty wallpapers that can be used as backgrounds to the mainPage of an application, by simply passing an integer value in the range [0,59] to the **background_image** parameter of the mainPage initialization function.

In case there is the need for a custom background image, any valid URL string can be passed in the background_image parameter. To customize the appearance of the background image (for instance adding some blur, contrast, etc.), the **background_filter** parameter can be set with a string containing the CSS image filters to apply (example: 'blur(2px) contrast(0.7)'). See https://developer.mozilla.org/en-US/docs/Web/CSS/filter for a list of available filters.

A similar example of a basic multi-page application can be found here: Multi Page Demo code

The corresponding running dashboard can be viewed here: Multi Page Demo

vois library

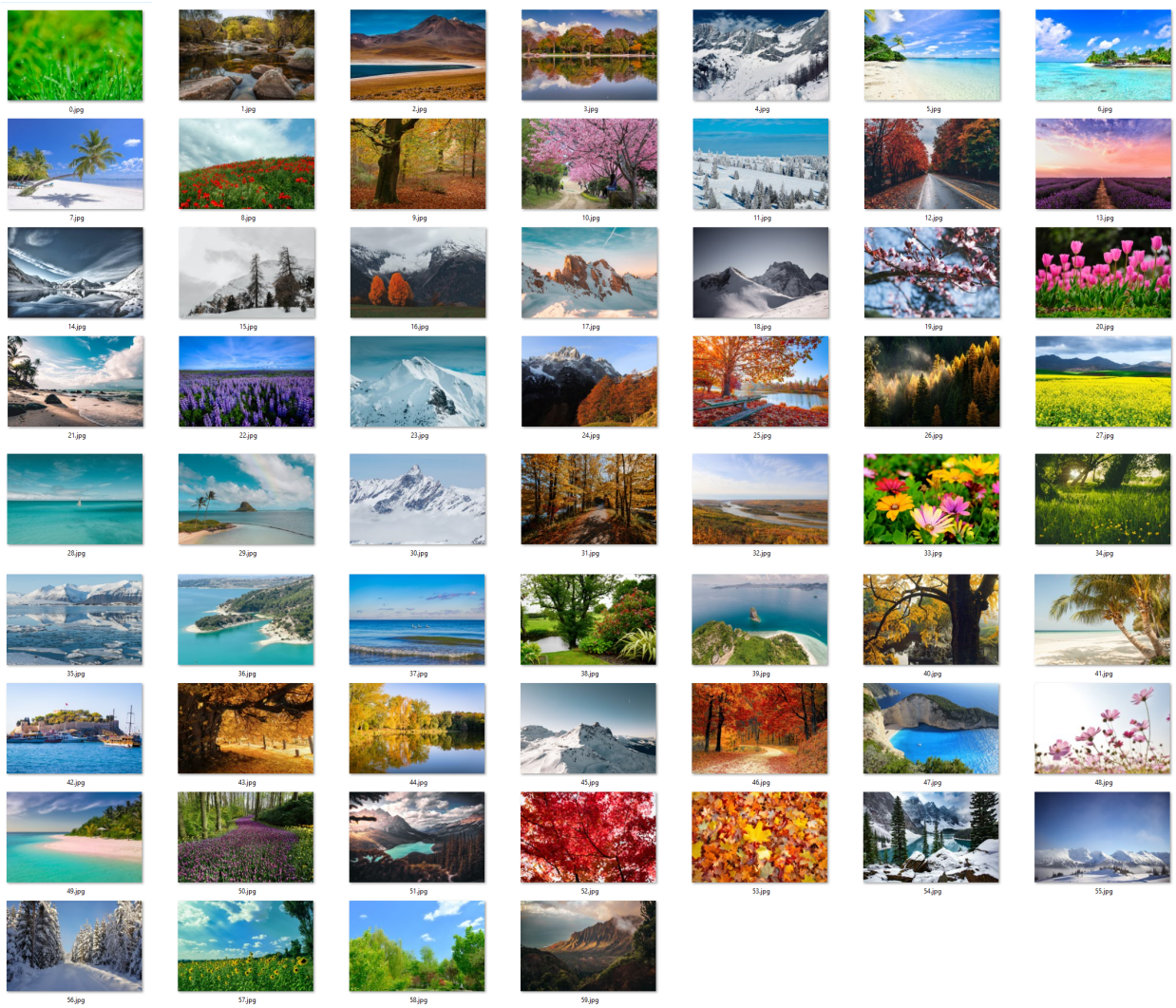


Fig. 2.19: Preview of the standard wallpapers provided by the vois library

CHAPTER

THREE

REFERENCE MANUAL

3.1 Setup

The vois library can be installed by executing: **pip install vois**

3.2 Packages

The vois library is grouped in these packages:

General package	Contains modules that define utilities functions and classes of general use (geojson, maps, svg, etc.)
Vuetify package	Contains modules that define classes to simplify the creation of GUI elements using ipyvuetify widgets

3.2.1 General package

The **General** package is made up of these modules:

colors	Utility functions and classes to manage colors and color interpolation.
download	Utility functions for downloading text and binary files
eucountries	Utility functions and classes to manage information on EU countries.
geojsonUtils	Utility functions to manage geospatial vector data in geojson format.
interMap	Utility functions for the creation of interactive maps using BDAP interactive library.
ipytrees	Utility functions for the creation ipytrees from hierarchical data.
leafletMap	Utility functions for the creation of interactive maps using ipyleaflet Map.
svgBarChart	SVG BarChart to display interactive vertical bars.
svgBubblesChart	SVG bubbles chart from a pandas DataFrame.
svgGraph	SVG visualization of a graph.
svgHeatmap	SVG heatmap chart from a pandas DataFrame.
svgMap	European map implemented in SVG.
svgPackedCirclesChart	SVG Packed Circles chart from a pandas DataFrame.
svgRankChart	SVG RankChart to display vertically aligned rectangles.
svgUtils	SVG drawings for general use.
textpopup	Map popup widget to display titles and texts in a geographic position on a ipyleaflet Map.
treemapPlotly	Utility functions to prepare data for Plotly Treemap, Sunburst and Icicle plots.
urlOpen	Utility functions to open a web page.
urlUpdate	Utility functions to update the URL of the page that launched the dashboard.

To use the modules of the **General** package they have to be imported using code like:

```
from vois import colors
print( colors.string2rgb('#ff00bb') )
```

3.2.2 Vuetify package

The **Vuetify** package is made up of these modules:

app	App class to easily define the structure of a typical Voilà dashboard.
basemaps	Widget to select the basemap to visualise on a ipyleaflet Map
button	Button widget to call a python function when clicked.
card	Simple card with title, subtitle and image.
cardsGrid	Cards with title, subtitle and image displayed in rows and columns
colorPicker	Input widget to select a color
datatable	Display of a Pandas DataFrame in a data-table widget.
datePicker	Input widget to select a date
dayCalendar	Calendar widget showing days with events

continues on next page

Table 3.1 – continued from previous page

dialogGeneric	Generic modal dialog-box to ask input from the user.
dialogMessage	Dialog-box to display a message for the user.
dialogWait	Dialog-box to display a message to the user during a lenghty operation.
dialogYesNo	Dialog-box to ask a yes-no question to the user.
fab	Floating-action-button to be displayed in absolute mode on the page.
fontsettings	Font settings for ipyvuetify widgets.
footer	Footer bar to be displayed at the bottom of a Voilà dashboard.
iconButton	Button displaying an icon.
label	Label widget to display a text with an optional icon.
layers	Widget to manage the layers Table Of Content for a ipyleaflet Map
mainPage	Initial page for an application
menu	Menu widget opened on hover on a button.
multiSwitch	Widget to select independent options using a list of buttons displayed horizontally or vertically.
page	Fullscreen page
paletteEditor	Widget for the creation and editing of color palettes.
palettePicker	Selection of a palette of colors
palettePickerEx	Extended selection of a palette of different families (sequential, divergent, etc.)
popup	Popup window opened at hover on a button.
progress	Circular progress bar to use for lenghty operations.
queryStrings	Read parameters passed in the URL of the Voila dashboard
radio	Radio buttons to allow users to select from a predefined set of options.
rangeSlider	Slider to select a range of numeric values.
rangeSliderFloat	Widget to select a range of float values
selectImage	Select widget that displays images and enables for single selection.
selectMultiple	Multiple selection widget.
selectSingle	Single selection widget from a dropdown list.
settings	General settings for ipyvuetify widgets.
sidePanel	Side panel that opens on the side of the screen to show content or get user input.
slider	Slider widget is a better visualization of the number input.
sliderFloat	Widget to select a float value
snackbar	Widget to display a quick message to the user in an overlapping window that will disappear after a timeout
sortableList	Vertically aligned list of customizable cards with items that can be moved, added and removed.
svgsGrid	Display and selection of a list of SVG files
switch	The switch widget provides users the ability to choose between two distinct values.
tabs	Widget to select among alternative display using a list of tabs displayed horizontally or vertically.
textlist	Widget to display text strings vertically aligned.

continues on next page

Table 3.1 – continued from previous page

title	Class that implements a title bar that can be used as a simple main interface for a dashboard.
toggle	Widget to select among alternative options using a list of buttons displayed horizontally or vertically.
tooltip	Add tooltip text to a widget: returns a "modified" widget to be used instead of the original one.
treeview	Simplified creation of v-treeview vuetify widget to display hierarchical data in a tree
upload	Widget to upload files from the user local machine

To use the modules of the **Vuetify** package they have to be imported using code like:

```
from vois.vuetify import app
a = app.app()
a.show()
```

3.3 Modules

3.3.1 General modules

The general modules of the vois library contain functions and classes of general use.

3.3.1.1 colors module

Utility functions and classes to manage colors and color interpolation.

class colors.**colorInterpolator**(*colorlist*, *minValue=0.0*, *maxValue=100.0*)

Class to perform color interpolation given a list of colors and a numerical range [minvalue,maxvalue]. The list of colors is considered as a linear range spanning from minvalue to maxvalue and the method *colors.colorInterpolator.GetColor()* can be used to calculate any intermediate color by passing as input any numeric value.

Parameters

- **colorlist** (*list of strings representing colors in 'rgb(r,g,b)' or '#rrggbb' format*) – Input list of colors
- **minvalue** (*float, optional*) – Minimum value for the interpolation (default is 0.0)
- **maxvalue** (*float, optional*) – Maximum numerical value for the interpolation (default is 100.0)

Examples

Creation of a color interpolator from a list of custom colors:

```
from vois import colors

colorlist = ['rgb(247,251,255)',
             'rgb(198,219,239)',
             'rgb(107,174,214)',
             'rgb(33,113,181)',
             'rgb(8,48,107)']
c = colors.colorInterpolator(colorlist)
print( c.GetColor(50.0) )
```

Creation of a color interpolator using one of the Plotly library predefined colorscales (see Plotly sequential color scales and Plotly qualitative color sequences):

```
import plotly.express as px
from vois import colors

c = colors.colorInterpolator(px.colors.sequential.Viridis, 0.0, 100.0)
print( c.GetColor(33.3) )
```

To visualize a color palette from a list of colors, the `colors.paletteImage()` function can be used:

```
from vois import colors
import plotly.express as px

img = colors.paletteImage(px.colors.sequential.Blues, width=400, height=40)
display(img)
```

Fig. 3.1: Display of a Plotly colorscale.

GetColor(*value*)

Returns a color in the '#rrggbb' format linearly interpolated in the [minvalue,maxvalue] range

Parameters

value (*float*) – Numeric value for which the color has to be calculated

Return type

A string containing the color represented as hexadecimals in the '#rrggbb' format

GetColors(*num_classes*)

Returns a list of colors in the '#rrggbb' format covering all the input colors

Parameters

num_classes (*int*) – Number of colors to interpolate

Return type

A list of strings containing the colors represented as hexadecimals in the '#rrggbb' format

`colors.darken(color1, color2)`

Returns the Darken blend of two colors strings

`colors.hex2rgb(color)`

Converts from a hexadecimal string representation of the color '#rrggbb' to a (r,g,b) tuple

Parameters

color (*string*) – A string containing the color represented as hexadecimals in the '#rrggbb' format

Return type

Tuple of 3 integers representing the RGB components in the range [0,255]

Example

Convert a color from '#rrggbb' to (r,g,b):

```
from vois import colors
print( colors.hex2rgb( '#ff0000' ) )
```

`colors.image2Base64(img)`

Given a PIL image, returns a string containing the image in base64 format

Parameters

img (*PIL image*) – Input PIL image

Return type

A string containing the image in base64 format

`colors.isColorDark(rgb)`

Returns True if the color (r,g,b) is dark

Parameters

rgb (*tuple*) – Tuple of 3 integers representing the RGB components in the range [0,255]

`colors.multiply(color1, color2)`

Returns the Multiply blend of two colors strings

`colors.paletteImage(colorlist, width=400, height=40, interpolate=True)`

Given a list of colors, calculates and returns a PIL image displaying the color palette.

Parameters

- **colorlist** (*list of strings representing colors in 'rgb(r,g,b)' or '#rrggbb' format*) – Input list of colors
- **width** (*int, optional*) – Width in pixel of the image (default is 400)
- **height** (*int, optional*) – Height in pixel of the image (default is 40)
- **interpolate** (*bool, optional*) – If True the colors of the list are interpolated, if False, only the color in the list are displayed (default is True)

Return type

A PIL image displaying the color palette

Examples

Creation of a color palette image from a list of colors:

```
from vois import colors
import plotly.express as px

img = colors.paletteImage(px.colors.sequential.Viridis, width=400, height=40)
display(img)
```

Fig. 3.2: Display of a Plotly colorscale.

`colors.randomColor()`

Returns a random color in the '#rrggbb' format

`colors.rgb2hex(rgb)`

Converts from a color represented as a (r,g,b) tuple to a hexadecimal string representation of the color '#rrggbb'

Parameters

rgb (*tuple of 3 int values*) – Input color described by its RGB components as 3 integer values in the range [0,255]

Return type

A string containing the color represented as hexadecimals in the '#rrggbb' format

Example

Convert a color from (r,g,b) to '#rrggbb':

```
from vois import colors
print( colors.rgb2hex( (255,0,0) ) )
```

`colors.string2rgb(s)`

Converts from string representation of the color 'rgb(r,g,b)' or '#rrggbb' to a (r,g,b) tuple

Parameters

color (*string*) – A string containing the color represented in the 'rgb(r,g,b)' format or in the '#rrggbb' format

Return type

Tuple of 3 integers representing the RGB components in the range [0,255]

`colors.text2rgb(color)`

Converts from string representation of the color 'rgb(r,g,b)' to a (r,g,b) tuple

Parameters

color (*string*) – A string containing the color represented in the 'rgb(r,g,b)' format

Return type

Tuple of 3 integers representing the RGB components in the range [0,255]

Example

Convert a color from 'rgb(r,g,b)' to (r,g,b):

```
from vois import colors
print( colors.text2rgb( 'rgb(255,0,0)' ) )
```

3.3.1.2 download module

Utility functions for downloading text and binary files

download.**downloadBytes**(*bytesobj*, *fileName='download.bin'*)

Download of an array of bytes as a binary file.

Parameters

- **bytesobj** (*bytearray or bytes object*) – Bytes to be written in the binary file to be downloaded
- **fileName** (*str, optional*) – Name of the file to download (default is "download.bin")

Example

In order to download a binary file, the Output widget download.output must be displayed inside the notebook. This is required because the download operation is based on the execution of Javascript code, and this requires an Output widget displayed. After the download.output widget is visible, then the download.downloadBytes function can be called:

```
from vois import download

display(download.output)

with download.output:
    download.downloadBytes(bytearray(b'ajgh lkjhl '))
```

download.**downloadText**(*textobj*, *fileName='download.txt'*)

Download of a string as a text file.

Parameters

- **textobj** (*str*) – Text to be written in the text file to be downloaded
- **fileName** (*str, optional*) – Name of the file to download (default is "download.txt")

Example

In order to download a text file, the Output widget download.output must be displayed inside the notebook. This is required because the download operation is based on the execution of Javascript code, and this requires an Output widget displayed. After the download.output widget is visible, then the download.downloadText function can be called:

```
from vois import download

display(download.output)

with download.output:
    download.downloadText('aaa bbb ccc')
```

3.3.1.3 eucountries module

Utility functions and classes to manage information on EU countries.

class eucountries.**countries**

Class to store information on all the EU countries.

Parameters

countries_list (*list of eucountries.country instances*) – List of countries in the European Union

Examples

Get the list of all European Union or Euro Area countries:

```
from vois import eucountries as eu

countriesEU = eu.countries.EuropeanUnion()
print(countriesEU)

countriesEuro = eu.countries.EuroArea()
print(countriesEuro)
```

Display the flag image of a country given its code:

```
from vois import eucountries as eu
display( eu.countries.byCode('IT').flagImage() )
```

Display the flag image of a country given its name:

```
from vois import eucountries as eu
display( eu.countries.byName('Lithuania').flagImage() )
```

Get the list of all the European Union country codes:

```
from vois import eucountries as eu
display( eu.countries.EuroAreaCodes() )
```

```
display( eu.countries.byName('Lithuania').flagImage() )
```

Fig. 3.3: Display of the flag of an EU country

classmethod EuroArea(*sortByName=True*)

Static method that returns the list of all the country belonging to the Euro Area

Parameters

sortByName (*bool, optional*) – If True, the returned list of countries is sorted by the name of the country (default is True)

classmethod EuroAreaCodes()

Static method that returns the list of all the iso2codes of the countries belonging to the Euro Area

classmethod EuropeanUnion(*sortByName=True*)

Static method that returns the list of all the country belonging to the European Union

Parameters

sortByName (*bool, optional*) – If True, the returned list of countries is sorted by the name of the country (default is True)

classmethod EuropeanUnionCodes()

Static method that returns the list of all the iso2codes of the countries belonging to the European Union

classmethod EuropeanUnionNames()

Static method that returns the list of all the names of the countries belonging to the European Union

classmethod add(*name, iso2code, euro=False, iscountry=True, population=0*)

Static method to add a country to the countries_list

Parameters

- **name** (*str*) – Name of the country
- **iso2code** (*str*) – Two letter code of the country defined by EUROSTAT (https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes)
- **euro** (*bool, optional*) – Flag that is True for countries belonging to the Euro Area (default is False)
- **iscountry** (*bool, optional*) – True if it is a real country (default is True). For instance 'Euro Area' and 'European Union' have this value set to False
- **population** (*int, optional*) – Last known population of the country

classmethod byCode(*iso2code*)

Static method that returns a country given its iso2code, or None if not existing

Parameters

iso2code (*str*) – Two letter code of the country defined by EUROSTAT (https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes)

Return type
Instance of *country* or None

`classmethod byName(name)`

Static method that returns a country given its name, or None if not existing

Parameters
name (*str*) – Name of the country

Return type
Instance of *country* or None

`class eucountries.country(name, iso2code, euro=False, iscountry=True, population=0)`

Class to store information on a single EU country.

Parameters

- **name** (*str*) – Name of the country
- **iso2code** (*str*) – Two letter code of the country defined by EUROSTAT (https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes)
- **iso3code** (*str*) – Three letter code of the country defined by ISO 3166 (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3)
- **euro** (*bool, optional*) – Flag that is True for countries belonging to the Euro Area (default is False)
- **iscountry** (*bool, optional*) – True if it is a real country (default is True). For instance 'Euro Area' and 'European Union' have this value set to False
- **population** (*int, optional*) – Last known population of the country

`flagImage()`

Returns the flag of the country as a PIL.Image object

`class eucountries.language(name, abbreviation, population=0)`

Class to store information on a language of the European Union.

Parameters

- **name** (*str*) – Name of the language
- **abbreviation** (*str*) – Abbreviation used for the language
- **population** (*int, optional*) – Last known population that uses the language

`class eucountries.languages`

Class to store information on all the European Union languages.

Parameters
languages_list (*list of eucountries.language instances*) – List of languages in the European Union

Examples

Print the list of all European Union languages:

```
from vois import eucountries as eu
names = eu.languages.EuropeanUnionLanguages(sortByName=False)
print(names)
```

Print the list of all European Union languages abbreviations:

```
from vois import eucountries as eu
abbrev = eu.languages.EuropeanUnionAbbreviations()
print(abbrev)
```

classmethod EuropeanUnionAbbreviations()

Static method that returns the list of all the abbreviations of the EU languages

classmethod EuropeanUnionLanguages(*sortByName=True*)

Static method that returns the list of all the languages of the European Union

Parameters

sortByName (*bool, optional*) – If True, the returned list of languages is sorted by the name of the language (default is True)

classmethod byAbbreviation(*abbreviation*)

Static method that returns a language given its abbreviation, or None if not existing

Parameters

abbreviation (*str*) – Two letter code of the language

Return type

Instance of language class or None

classmethod byName(*name*)

Static method that returns a language given its name, or None if not existing

Parameters

name (*str*) – Name of the language

Return type

Instance of language class or None

3.3.1.4 geojsonUtils module

Utility functions to manage geospatial vector data in geojson format.

geojsonUtils.**geojsonAll**(*geojson*, *attributeName*)

Given a geojson string, returns the list of values of the attribute attributeName for all the features

Parameters

- **geojson** (*str*) – String containing data in geojson format
- **attributeName** (*str*) – Name of one of the attributes

Return type

List containing the values of the attribute for all the features of the input geojson dataset

Example

Load a geojson from file and print all the values of one if its attributes:

```
from vois import geojsonUtils

geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

print('Values = ', geojsonUtils.geojsonAll(geojson,'ha'))
```

```
from voilalibrary import geojsonUtils

geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

print('Values = ', geojsonUtils.geojsonAll(geojson,'ha'))
```

```
Values =  [6.73583984375, 0.926895022392, 0.617895007133, 0.505702018738, 7.41506004333, 0.0679828971624, 7.862549781
8, 0.711094021797, 0.957135021687, 1.32553005219, 4.52396011353, 0.327562004328, 1.03611004353, 0.765770971775, 1.809
57996845, 7.11729001999, 0.672756016254, 3.89577007294, 1.50476002693, 0.737752020359, 2.31732988358, 1.23739004135,
1.20881998539, 1.11463999748, 3.2857298851, 0.780149996281, 0.794167995453, 0.318890988827, 0.845695018768, 5.9939999
5804, 0.29587700963, 3.60988998413, 1.26472997665, 0.468057990074, 7.30466985703, 1.48009002209, 0.267468005419, 0.23
2388004661, 0.250239998102, 4.457459926610001, 0.0160592999309, 0.481920987368, 1.66877996922, 0.361173003912, 5.2171
6022491, 0.489280998707, 0.771668970585, 2.21916007996, 0.504687011242, 0.115186996758]
```

Fig. 3.4: Read a geojson file and print the values of one of its attributes for all the features of the dataset

`geojsonUtils.`**`geojsonAttributes`**`(`*geojson*`)`

Given a geojson string, returns the list of the attribute names of the features

Parameters

geojson (*str*) – String containing data in geojson format

Return type

List of strings containing the names of the attributes of the features in the input geojson string

Example

Load a geojson from file and print the names of its attributes:

```
from vois import geojsonUtils

geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

print('Attributes = ', geojsonUtils.geojsonAttributes(geojson))
```

```
from voilalibrary import geojsonUtils

geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

print('Attributes = ', geojsonUtils.geojsonAttributes(geojson))
```

```
Attributes =  ['seca_regad', 'municipi', 'comarca', 'cultiu', 'id', 'id_mun', 'campanya', 'grup', 'ha', 'provincia']
```

Fig. 3.5: Read a geojson file and print the names of its attributes

`geojsonUtils.`**`geojsonCount`**`(`*geojson*`)`

Given a geojson string, returns the number of features

Parameters
geojson (*str*) – String containing data in geojson format

Return type
Integer corresponding to the number of features in the input geojson string

`geojsonUtils.geojsonFilter(geojson, fieldname, fieldvalue)`

Filter a geojson by keeping only the features for which <fieldname> has value <fieldvalue> (fieldvalue can be also a list)

Parameters

- **geojson** (*str*) – String containing data in geojson format
- **fieldname** (*str*) – Name of one of the attributes of the input geojson
- **fieldvalue** (*single value or list of values*) – Comparison value. Only the input features having this value on the <fieldname> attribute are kept in the output geojson returned

Return type
a string containing the modified geojson containing only the features that pass the filter operation

`geojsonUtils.geojsonJoin(geojson, keyname, addedfieldname, keytovaluedict, innerMode=False)`

Add a field to a geojson by joining a python dictionary through match with the field named keyname. If innerMode is True, the output geojson will only keep the joined features, otherwise all the original features are returned

Parameters

- **geojson** (*str*) – String containing data in geojson format
- **keyname** (*str*) – Name of the attribute of the input geojson to use as internal key for the join operation
- **addedfieldname** (*str*) – Name of the attribute to add to the input geojson as a result of the join operation
- **keytovaluedict** (*dict*) – Dictionary (key-value pairs) to use as joined values. The keys of the <keytovaluedict> are used as foreign keys to match the values of the <keyname> attribute of the input geojson. When a match is found, the attribute <addedfieldname> is added to the corresponding feature having the value read from the <keytovaluedict>
- **innerMode** (*bool, optional*) – If innerMode is True, the output geojson will only keep the successfully joined features, otherwise all the original features are returned

Return type
a string containing the modified geojson after the join operation

Example

Load a geojson from file, print some information on attributes and values of the features, then join the features with a dictionary:

```
from vois import geojsonUtils

# Load a geojson file
geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

# Add a new field by joining with a dictionary (with innerMode flag set to True)
keytovalue = { 37661: 'aaa', 37662: 'bbb'}
geojsonnew = geojsonUtils.geojsonJoin(geojson,'id', 'value', keytovalue,␣
```

(continues on next page)

(continued from previous page)

```
↪innerMode=True)

# Print the 'value' attribute values for the joined geojson dataset
print('Joined values =', geojsonUtils.geojsonAll(geojsonnew,'value'))
```

```
from voilalibrary import geojsonUtils

# Load a geojson file
geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

# Add a new field by joining with a dictionary (with innerMode flag set to True)
keytovalue = { 37661: 'aaa', 37662: 'bbb'}
geojsonnew = geojsonUtils.geojsonJoin(geojson,'id', 'value', keytovalue, innerMode=True)

# Print the 'value' attribute values for the joined geojson dataset
print('Joined values =', geojsonUtils.geojsonAll(geojsonnew,'value'))
```

```
Joined values = ['aaa', 'bbb']
```

Fig. 3.6: Result of the Join operation

`geojsonUtils.`**`geojsonJson`**(*geojson*)

Given a geojson string, returns a json dictionary after having tested that the input string contains a valid geojson

Parameters

geojson (*str*) – String containing data in geojson format

Return type

Json dictionary

Raises

Exception if the input string is not in geojson format –

`geojsonUtils.`**`geojsonLoadFile`**(*filepath*)

Load a geojson content from file, testing that it contains valid geojson data

Parameters

filepath (*str*) – File path of the geojson file to load

Return type

File content as a geojson string

Example

Load a geojson from file, print the geojson string:

```
from vois import geojsonUtils

geojson = geojsonUtils.geojsonLoadFile('./data/example.geojson')

print(geojson)
```

```
print(geojson)
```


```
{"type": "FeatureCollection", "name": "maize50", "crs": {"type": "name", "properties": {"name":
"urn:ogc:def:crs:EPSG::25831"}}, "features": [{"type": "Feature", "properties": {"id": 37661, "ca
mpanya": 2019.0, "provincia": "Barcelona", "comarca": "Osona", "municipi": "GURB", "id_mun": "080
99", "grup": "CEREALS D'ESTIU", "cultiu": "BLAT DE MORO", "seca_regad": "S", "ha": 6.7358398437
5}, "geometry": {"type": "MultiPolygon", "coordinates": [[[[439503.72860000003, 4647271.90080000
1], [439504.69319999963, 4647266.9131000005], [439509.40139999986, 4647262.759199999], [439509.60
02000002, 4647261.640699999], [439502.2701000003, 4647246.6252], [439495.1889000004, 4647233.915
8], [439483.3816999998, 4647215.9416000005], [439474.4846000001, 4647204.5042], [439467.587799999
86, 4647196.3353], [439456.20419999957, 4647183.2532], [439449.9068999998, 4647174.530099999], [4
39445.06240000017, 4647166.781099999], [439438.2734000003, 4647156.6009], [439431.49760000035, 46
47147.1559], [439428.5892000003, 4647143.2809], [439420.35059999954, 4647135.775], [439412.355499
9996, 4647129.960899999], [439405.57579999976, 4647126.5657], [439399.1935999999, 4647125.0101],
[439394.0301999999, 4647127.0923999995], [439381.5999999996, 4647127.319499999], [439367.30790000
036, 4647126.659700001], [439355.8722000001, 4647126.217499999], [439336.6255000001, 4647128.5226
00001], [439324.53170000017, 4647134.796599999], [439321.0111999996, 4647137.3211], [439319.69299
999997, 4647140.729499999], [439319.4742999999, 4647143.5888], [439317.93400000036, 4647144.579
2], [439315.5148, 4647145.1303], [439312.87739999965, 4647145.348300001], [439308.37119999994, 46
47144.6888], [439301.4430999998, 4647144.2546999995], [439293.7413999997, 4647142.7104], [439292.
86340000015, 4647138.3155000005], [439294.84530000016, 4647135.348099999], [439292.86149999965, 4
```


Fig. 3.7: Read a geojson file and print its content

3.3.1.5 interMap module

Utility functions for the creation of interactive maps using BDAP interactive library.

`interMap.bivariateLegend`(*v*, *filters1*, *filters2*, *colorlist1*, *colorlist2*, *title=''*, *title1=''*, *title2=''*, *names1=[]*, *names2=[]*, *fontsize=14*, *fontweight=400*, *stroke='#000000'*, *stroke_width=0.25*, *side=100*, *resizewidth=''*, *resizeheight=''*)

Creation of a bivariate choropleth legend for a polygon vector layer. See Bivariate Choropleth Maps: A How-to Guide for the idea. The function creates a legend for vector layer v based on two attributes of the layer and returns a string containing the SVG representation of the legend (that can be displayed using display(HTML(svgstring) call)

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it is not portable in other environments! Please refer to the module `lefletMap` for geospatial function not related to BDAP.

Parameters

- **v** (*instance of inter.VectorLayer class*) – Vector layer instance for which the bivariate legend has to be built
- **filters1** (*list of strings*) – List of strings defining the conditions for the classes based on the first attribute
- **filters2** (*list of strings*) – List of strings defining the conditions for the classes based on the second attribute
- **colorlist1** (*list of colors*) – List of colors to use for the legend on the first attribute (see Plotly sequential color scales and Plotly qualitative color sequences)

- **colorlist2** (*list of colors*) – List of colors to use for the legend on the second attribute (see Plotly sequential color scales and Plotly qualitative color sequences)
- **title** (*str, optional*) – Main title of the legend chart (default is ‘’)
- **title1** (*str, optional*) – Title for the legend on the first attribute. It will be displayed vertically in the Y axis of the SVG. Default is ‘’.
- **title2** (*str, optional*) – Title for the legend on the second attribute. It will be displayed horizontally in the X axis of the SVG. Default is ‘’.
- **names1** (*list of strings, optional*) – List containing one string for each of the classes of the legend on the first attribute (default is [])
- **names2** (*list of strings, optional*) – List containing one string for each of the classes of the legend on the second attribute (default is [])
- **fontsize** (*int, optional*) – Size in pixels of the font used for texts (default is 14)
- **fontweight** (*int, optional*) – Weight of the font used for title texts (default is 400)
- **stroke** (*str, optional*) – Color to use for the border of the polygons (default is ‘#000000’)
- **stroke_width** (*float, optional*) – Width in pixels of the stroke to use for the border of the polygons (default is 0.25)
- **side** (*int, optional*) – Side in pixels of the squares displayed in the SVG legend (default is 100)
- **resizewidth** (*str, optional*) – Width of the resizing container (default is ‘’)
- **resizeheight** (*str, optional*) – height of the resizing container (default is ‘’)

Return type

a string containing SVG text to display the bivariate legend using a call to display(HTML(...))

Example

Creation of a bivariate choropleth legend for the polygons of the Italian provinces. The first attribute is the short name of the province (attribute ‘SIGLA’), and the second attribute is the SHAPE_AREA attribute which contains the dimension in squared meters:

```
from ipywidgets import widgets, Layout, HTML
from IPython.display import display

from jeodpp import inter, imap
from vois import interMap, geojsonUtils

# Load data on italian provinces
geojson = geojsonUtils.geojsonLoadFile('./data/ItalyProvinces.geojson')
vector = interMap.interGeojsonToVector(geojson)
vector = vector.parameter("identifyfield", "SIGLA DEN_PROV SHAPE_AREA")
vector = vector.parameter("identifyseparator", "<br>")

# Create and display a Map instance
m = imap.Map(basemap=1, layout=Layout(height='600px'))
display(m)
```

(continues on next page)

(continued from previous page)

```
# Creation of the bivariate legend
colorlist1 = ['#f3f3f3', '#eac5dd', '#e6a3d0']
colorlist2 = ['#f3f3f3', '#c2f1d5', '#8be2ae']

svg = interMap.bivariateLegend(vector,
                               ["[SIGLA] < 'FE'", "[SIGLA] >= 'FE' and [SIGLA] <=
↪'PU'", "[SIGLA] > 'PU'"],
                               ["[SHAPE_AREA] < 2500000000", "[SHAPE_AREA] >=␣
↪2500000000 and [SHAPE_AREA] <= 4500000000", "[SHAPE_AREA] > 4500000000 and [SHAPE_
↪AREA] <= 7500000000", '[SHAPE_AREA] > 7500000000'],
                               colorlist1,
                               colorlist2,
                               title='Example of Bivariate Choropleth',
                               title1="Province initials",
                               names1=['< FE', 'in [FE,PU]', '> PU'],
                               title2="Province area",
                               names2=['Small', 'Medium', 'Large', 'XLarge'],
                               fontsize=24,
                               fontweight=500)

# Display of the vector layer on the map
p = vector.process()
m.clear()
m.addLayer(p.toLayer())
m.zoomToImageExtent(p)

inter.identifyPopup(m,p)

# Display the bivariate choropleth legend
display(HTML(svg))
```

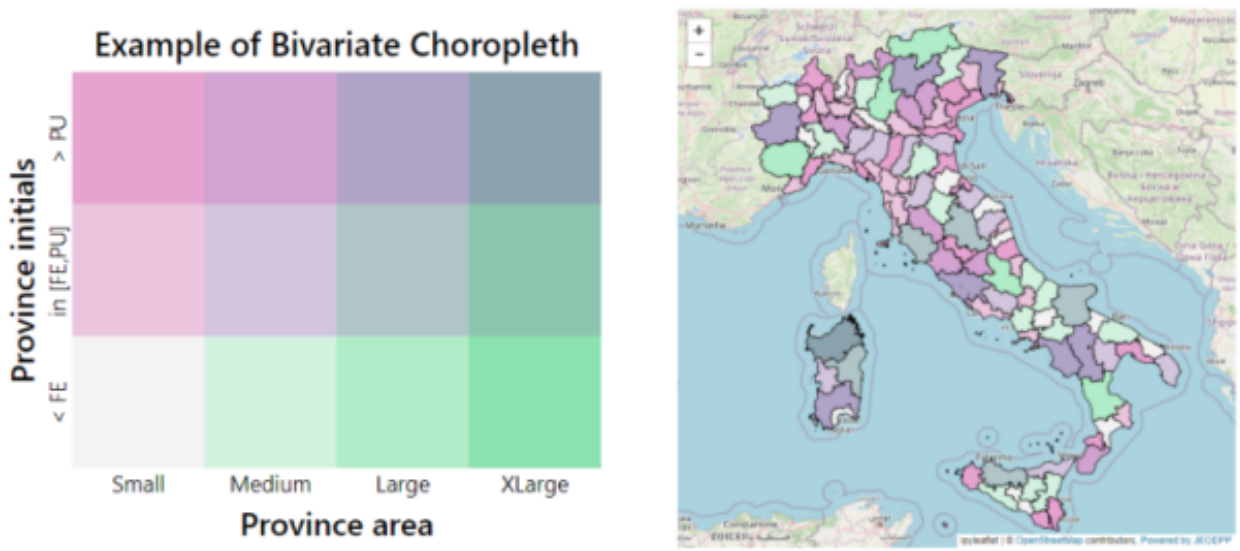


Fig. 3.8: Example of an interactive map showing polygons colored with a bivariate choropleth legend.

interMap.**countriesMap**(*df*, *code_column=None*, *value_column='value'*, *codes_selected=[]*, *center=None*, *zoom=None*, *width='99%'*, *height='400px'*, *min_width=None*, *basemap=1*, *colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921']*, *stdevnumber=2.0*, *stroke='#232323'*, *stroke_selected='#00ffff'*, *stroke_width=1.0*, *decimals=2*, *minallowed_value=None*, *maxallowed_value=None*)

Creation of an interactive map to display the countries of the world. An input Pandas DataFrame df is used to join a column of numeric values to the countries, using the iso2code (ISO 3166-2) as internal key attribute. Once the values are assigned to the countries, a graduated legend is calculated based on mean and standard deviation of the assigned values. A input list of colors is used to represent the countries given their assigned value.

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it is not portable in other environments! Please refer to the module `leafletMap` for geospatial function not related to BDAP.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to the countries. It has to contain at least a column with numeric values.
- **code_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the unique code of the countries in the ISO-3166-2 standard. This column is used to perform the join with the internal attribute of the countries vector dataset that contains the country code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the countries using the join on the ISO-3166-2 codes (default is 'value')
- **codes_selected** (*list of strings, optional*) – List of codes of countries to display as selected (default is [])
- **center** (*tuple of (lat,lon), optional*) – Geographical coordinates of the initial center of the interactive map visualization (default is None)
- **zoom** (*int, optional*) – Initial zoom level of the interactive map (default is None)
- **width** (*str, optional*) – Width of the map widget to create (default is '99%')
- **height** (*str, optional*) – Height of the map widget to create (default is '400px')
- **min_width** (*str, optional*) – Minimum width of the layout of the map widget (default is None)
- **basemap** (*int, optional*) – Basemap to use as background in the map visualization (default is 1). Valid values are in [1,39], see https://jeodpp.jrc.ec.europa.eu/services/processing/interhelp/3.2_map.html?highlight=basemap#inter.Map.printAvailableBasemaps for details
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)
- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to country polygons and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0

- **stroke** (*str, optional*) – Color to use for the border of countries (default is '#232323')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected countries (default is '#00ffff')
- **stroke_width** (*float, optional*) – Width of the border of the country polygons in pixels (default is 1.0)
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors

Return type
a jeodpp.imap instance (a Map object derived from the ipyleaflet Map)

Example

Creation of a map displaying a random variable on 4 european countries. The numerical values assigned to each of the countries are randomly generated using numpy.random.uniform and saved into a dictionary having the country code as the key. This dict is transformed to a Pandas DataFrame with 4 rows and having 'iso2code' and 'value' as columns. The graduated legend is build using the 'inverted' Reds Plotly colorscale (low values are dark red, intermediate values are red, high values are white):

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import interMap

countries = ['DE', 'ES', 'FR', 'IT']

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

m = interMap.countriesMap(df,
                          code_column='iso2code',
                          height='400px',
                          stroke_width=1.5,
                          stroke_selected='yellow',
                          colorlist=px.colors.sequential.Reds[::-1],
                          codes_selected=['IT'])
display(m)
```

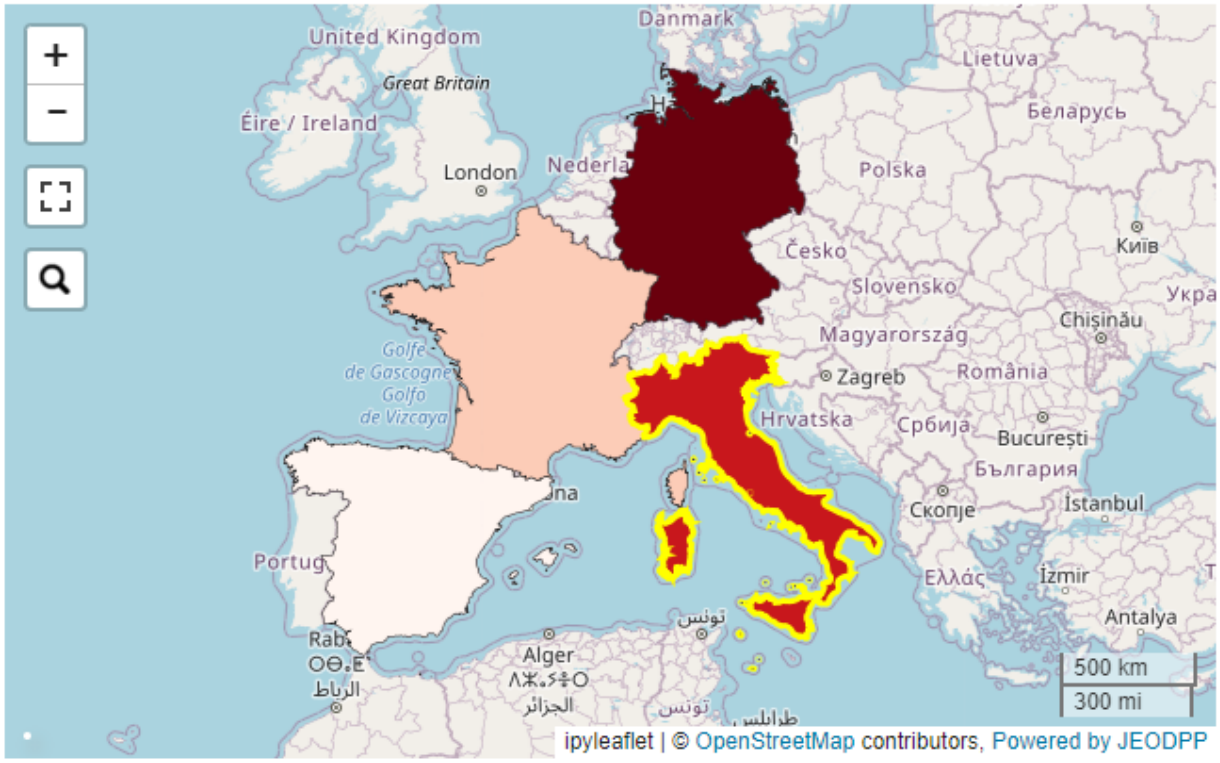


Fig. 3.9: Example of an interactive map displaying 4 european countries.

`interMap.geojsonMap`(*df*, *geojson_path*, *geojson_attribute*, *code_column=None*, *value_column='value'*, *codes_selected=[]*, *center=None*, *zoom=None*, *width='99%'*, *height='400px'*, *min_width=None*, *basemap=1*, *colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921']*, *stdevnumber=2.0*, *stroke='#232323'*, *stroke_selected='#00ffff'*, *stroke_width=1.0*, *decimals=2*, *minallowed_value=None*, *maxallowed_value=None*)

Creation of an interactive map to display a custom geojson dataset. An input Pandas DataFrame df is used to join a column of numeric values to the geojson features, using the <geojson_attribute> as the internal key attribute. Once the values are assigned to the features, a graduated legend is calculated based on mean and standard deviation of the assigned values. A input list of colors is used to represent the featuress given their assigned value.

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it is not portable in other environments! Please refer to the module `lefletMap` for geospatial function not related to BDAP.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to features. It has to contain at least a column with numeric values.
- **geojson_path** (*str*) – Path of the geojson file to load that contains the geographic features in geojson format
- **geojson_attribute** (*str*) – Name of the attribute of the geojson dataset that contains the unique codes of the features. This attribute will be use as internal key in the join operation

with the df Pandas DataFrame

- **code_column** (*str, optional*) – Name of the column of the df Pandas DataFrame containing the unique code of the features. This column is used to perform the join with the internal attribute of the geojson vector dataset that contains the unique code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the features using the join on geojson unique codes (default is 'value')
- **codes_selected** (*list of strings, optional*) – List of codes of features to display as selected (default is [])
- **center** (*tuple of (lat,lon), optional*) – Geographical coordinates of the initial center of the interactive map visualization (default is None)
- **zoom** (*int, optional*) – Initial zoom level of the interactive map (default is None)
- **width** (*str, optional*) – Width of the map widget to create (default is '99%')
- **height** (*str, optional*) – Height of the map widget to create (default is '400px')
- **min_width** (*str, optional*) – Minimum width of the layout of the map widget (default is None)
- **basemap** (*int, optional*) – Basemap to use as background in the map visualization (default is 1). Valid values are in [1,39], see https://jeodpp.jrc.ec.europa.eu/services/processing/interhelp/3.2_map.html?highlight=basemap#inter.Map.printAvailableBasemaps for details
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)
- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to features and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **stroke** (*str, optional*) – Color to use for the border of countries (default is '#232323')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected countries (default is '#00ffff')
- **stroke_width** (*float, optional*) – Width of the border of the country polygons in pixels (default is 1.0)
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors

Return type

a jeodpp.imap instance (a Map object derived from the ipyleaflet Map)

Example

Creation of a map displaying a custom geojson. The numerical values assigned to each of the countries are randomly generated using numpy.random.uniform and saved into a dictionary having the country code as the key. This dict is transformed to a Pandas DataFrame with 4 rows and having 'iso2code' and 'value' as columns. The graduated legend is build using the 'inverted' Reds Plotly colorscale (low values are dark red, intermediate values are red, high values are white):

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import interMap

countries = ['DE', 'ES', 'FR', 'IT']

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

m = interMap.geojsonMap(df,
                        './data/ne_50m_admin_0_countries.geojson',
                        'ISO_A2_EH',   # Internal attribute used as key
                        code_column='iso2code',
                        height='400px',
                        stroke_width=1.5,
                        stroke_selected='yellow',
                        colorlist=px.colors.sequential.Reds[::-1],
                        codes_selected=['IT'])
display(m)
```

interMap.**interGeojsonToVector**(*geojson*)

Load a geojson string and returns a inter.VectorLayer object (see https://jeodpp.jrc.ec.europa.eu/services/processing/interhelp/3.5_vectorlayer.html)

Parameters

geojson (*str*) – String containing data in geojson format

Return type

An instance of the inter.Vector class of the interapro library

---

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it in not portable in other environments!

---

interMap.**trivariateLegend**(*v*, *filter1*, *filter2*, *filter3*, *color1='#ff60ff'*, *color2='#fff60'*, *color3='#60fff'*, *color4='#ffffff'*, *title=''*, *title1=''*, *title2=''*, *title3=''*, *fontsize=14*, *fontweight=300*, *stroke='#000000'*, *stroke_width=0.25*, *radius=100*)

Creation of a trivariate choropleth legend for a polygon vector layer. See Some Thoughts on Multivariate Maps for the idea. The function creates a legend for vector layer v based on three attributes of the layer and returns a string containing the SVG representation of the legend (that can be displayed using display(HTML(svgstring) call).

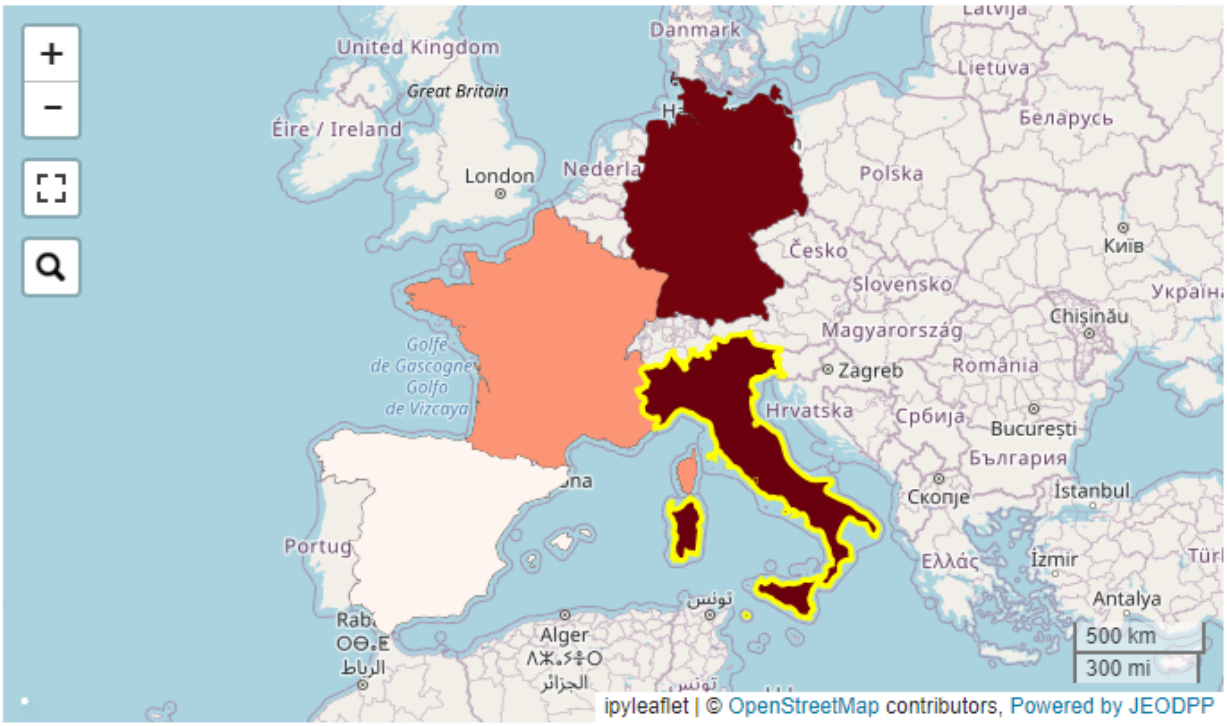


Fig. 3.10: Example of an interactive map displaying 4 european countries from a custom geojson file.

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it is not portable in other environments! Please refer to the module `lefletMap` for geospatial function not related to BDAP.

Parameters

- **v** (*instance of inter.VectorLayer class*) – Vector layer instance for which the trivariate legend has to be built
- **filter1** (*str*) – Condition to filter the polygons on the first attribute
- **filter2** (*str*) – Condition to filter the polygons on the second attribute
- **filter3** (*str*) – Condition to filter the polygons on the third attribute
- **color1** (*str, optional*) – Color to assign to polygons that satisfy the condition on the first attribute (default is '#ff80ff')
- **color2** (*str, optional*) – Color to assign to polygons that satisfy the condition on the second attribute (default is '#ffff80')
- **color3** (*str, optional*) – Color to assign to polygons that satisfy the condition on the third attribute (default is '#ff80ff')
- **color4** (*str, optional*) – Color to assign to polygons that do not satisfy any of the three conditions (default is '#ffffff')
- **title** (*str, optional*) – Main title of the legend chart (default is '')
- **title1** (*str, optional*) – Title for the legend on the first attribute (default is '')

- **title2** (*str, optional*) – Title for the legend on the second attribute (default is ‘’)
- **title3** (*str, optional*) – Title for the legend on the third attribute (default is ‘’)
- **fontsize** (*int, optional*) – Size in pixels of the font used for texts (default is 14)
- **fontweight** (*int, optional*) – Weight of the font used for title texts (default is 400)
- **stroke** (*str, optional*) – Color to use for the border of the polygons (default is ‘#000000’)
- **stroke_width** (*float, optional*) – Width in pixels of the stroke to use for the border of the polygons (default is 0.25)
- **radius** (*int, optional*) – Radius in pixels of the circles displayed in the SVG legend (default is 100)

Return type

a string containing SVG text to display the trivariate legend using a call to display(HTML(...))

Example

Creation of a simple trivariate choropleth legend (with 7 colors) for a polygons layer containing crop data. The three attributes AL_PERC, PC_PERC and PG_PERC contain the percentage presence of the three specific crops inside the polygon:

```
from ipywidgets import widgets, Layout, HTML
from IPython.display import display

from jeodpp import inter, imap
from vois import interMap

# Load data
vector = inter.loadLocalVector("DEBY_2019_LandCover.shp")
vector = vector.parameter("identifyfield", "LAU_NAME YEAR AL_PERC PC_PERC PG_PERC")
vector = vector.parameter("identifyseparator", "<br>")

# Create and display a Map instance
m = imap.Map(basemap=60, layout=Layout(height='600px'))
display(m)

# Creation of the bivariate legend
svg = interMap.trivariateLegend(vector,
                                "[AL_PERC] > 60",
                                "[PC_PERC] > 10",
                                "[PG_PERC] > 20",
                                '#ff60ff',
                                '#ffff60',
                                '#60ffff',
                                '#ffffff55',
                                title='Example of Trivariate Choropleth',
                                title1="Arable Land",
                                title2="Perm. Crop",
                                title3="Permanent Grassland",
                                fontsize=12,
                                fontweight=500,
```

(continues on next page)

(continued from previous page)

```
                                    radius=70)

# Display of the vector layer on the map
p = vector.process()
m.clear()
m.addLayer(p.toLayer())
m.zoomToImageExtent(p)

inter.identifyPopup(m,p)

# Display the trivariate choropleth legend
display(HTML(svg))
```

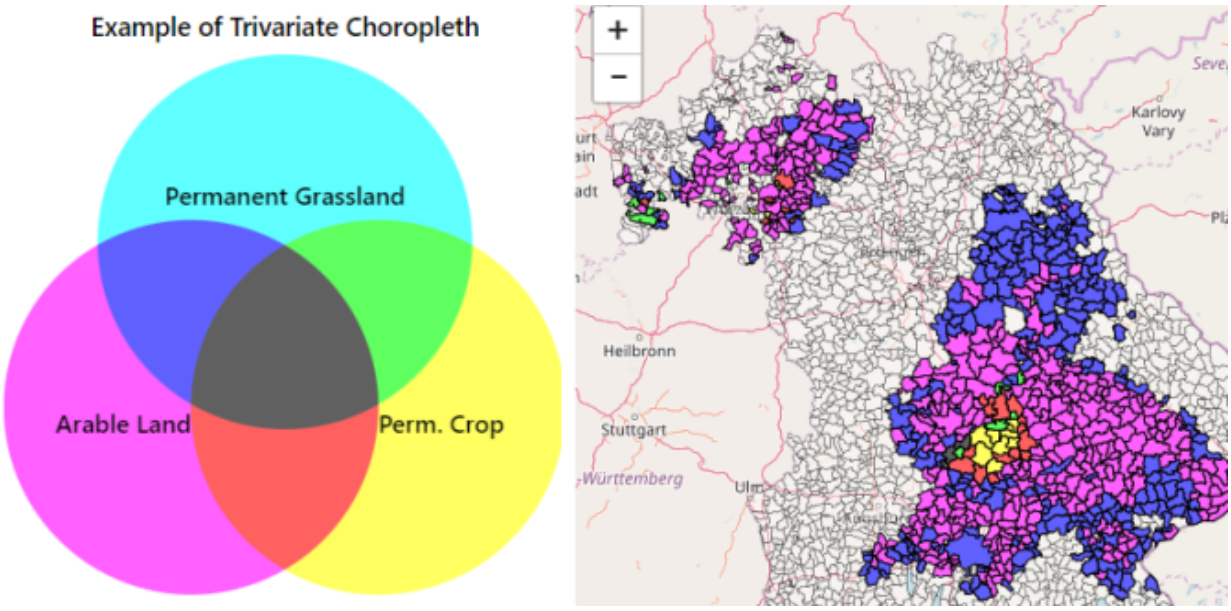


Fig. 3.11: Example of an interactive map showing polygons colored with a trivariate choropleth legend.

`interMap.`**`trivariateLegendEx`**(*v*, *attribute1*, *attribute2*, *attribute3*, *n=3*, *min1=0.0*, *max1=100.0*, *min2=0.0*, *max2=100.0*, *min3=0.0*, *max3=100.0*, *color1='#ff80f7'*, *color2='#00d1d0'*, *color3='#cfb000'*, *color4='#ffffff'*, *title=''*, *title1=''*, *title2=''*, *title3=''*, *fontsize=14*, *fontweight=400*, *stroke='#000000'*, *stroke_width=0.25*, *side=200*, *resizewidth=''*, *resizeheight=''*, *digits=2*, *maxticks=0*, *showarrows=True*)

Creation of a trivariate choropleth legend for a polygon vector layer. See Choropleth maps with tricolore for the idea. The function creates a legend for vector layer v based on three attributes of the layer and returns a string containing the SVG representation of the legend in the form of a triangle (that can be displayed using display(HTML(svgstring) call)

Note: This function is built on top of the BDAP interapro library to display dynamic geospatial dataset. For this reason it is not portable in other environments! Please refer to the module `lefletMap` for geospatial function not related to BDAP.

Parameters

- **v** (*instance of inter.VectorLayer class*) – Vector layer instance for which the trivariate legend has to be built
- **attribute1** (*str*) – Name of the first numerical attribute
- **attribute2** (*str*) – Name of the second numerical attribute
- **attribute3** (*str*) – Name of the third numerical attribute
- **n** (*int, optional*) – Number of intervals for each of the three numerical attributes (default is 3). Accepptable values are those in the range [2,10]
- **min1** (*float, optional*) – Minimum value for the first attribute (default is 0.0)
- **max1** (*float, optional*) – Maximum value for the first attribute (default is 100.0)
- **min2** (*float, optional*) – Minimum value for the second attribute (default is 0.0)
- **max2** (*float, optional*) – Maximum value for the second attribute (default is 100.0)
- **min3** (*float, optional*) – Minimum value for the third attribute (default is 0.0)
- **max3** (*float, optional*) – Maximum value for the third attribute (default is 100.0)
- **color1** (*str, optional*) – Color to assign to polygons that have the maximum value on the first attribute (default is '#ff80f7')
- **color2** (*str, optional*) – Color to assign to polygons that have the maximum value on the second attribute (default is '#00d1d0')
- **color3** (*str, optional*) – Color to assign to polygons that have the maximum value on the third attribute (default is '#cfb000')
- **color4** (*str, optional*) – Color to assign to polygons that have all the three values of the attributes smaller than the corresponding minimal value (default is '#ffffff'). For this color, the transparency can be set, for instance using '#ffffff88' for partial transparency or '#ffffffff' for full transparency.
- **title** (*str, optional*) – Main title of the legend chart (default is '')
- **title1** (*str, optional*) – Title for the legend on the first attribute. It will be displayed on the bottom side of the triangle SVG. Default is ''.
- **title2** (*str, optional*) – Title for the legend on the second attribute. It will be displayed on the right side of the triangle SVG. Default is ''.
- **title3** (*str, optional*) – Title for the legend on the third attribute. It will be displayed on the left side of the triangle SVG. Default is ''.
- **fontsize** (*int, optional*) – Size in pixels of the font used for texts (default is 14)
- **fontweight** (*int, optional*) – Weight of the font used for title texts (default is 400)
- **stroke** (*str, optional*) – Color to use for the border of the polygons (default is '#000000')
- **stroke_width** (*float, optional*) – Width in pixels of the stroke to use for the border of the polygons (default is 0.25)
- **side** (*int, optional*) – Dimension in pixels of one side of the triangle displayed in the SVG legend (default is 200)
- **resizewidth** (*str, optional*) – Width of the resizing container (default is '')
- **resizeheight** (*str, optional*) – height of the resizing container (default is '')

- **digits** (*int, optional*) – Number of decimal digits to use for displaying numerical values on the axis of the SVG chart (default is 2)
- **maxticks** (*int, optional*) – Maximum number of tick marks to display on each of the triangle sides. If 0 or less, the ticks for all the intervals are shown. Default is 0
- **showarrows** (*bool, optional*) – If True displays small arrows to help identify the three axes (default is True)

Return type
a string containing SVG text to display the trivariate legend using a call to display(HTML(...))

Example

Creation of a complex trivariate choropleth legend for a polygons layer containing crop data. The three attributes AL_PERC, PC_PERC and PG_PERC contain the percentage presence of the three specific crops inside the polygon:

```
from ipywidgets import widgets, Layout, HTML
from IPython.display import display

from jeodpp import inter, imap
from vois import interMap

# Load data
vector = inter.loadLocalVector("DEBY_2019_LandCover.shp")
vector = vector.parameter("identifyfield", "LAU_NAME YEAR AL_PERC PC_PERC PG_PERC")
vector = vector.parameter("identifyseparator", "<br>")

# Create and display a Map instance
m = imap.Map(basemap=60, layout=Layout(height='600px'))
display(m)

svg = interMap.trivariateLegendEx(vector,
                                  "AL_PERC",
                                  "PC_PERC",
                                  "PG_PERC",
                                  6,
                                  0.0,
                                  100.0,
                                  0.0,
                                  100.0,
                                  0.0,
                                  100.0,
                                  color1='#ff80f7',
                                  color2='#00d1d0',
                                  color3='#cfb000',
                                  color4='#ffffff00',
                                  title='Complex Trivariate Choropleth',
                                  title1="Arable Land",
                                  title2="Perm. Crop",
                                  title3="Perm. Grassl.",
                                  fontsize=18,
                                  fontweight=500,
```

(continues on next page)

(continued from previous page)

```
                                    side=400,
                                    digits=0,
                                    maxticks=5,
                                    showarrows=True)

# Display of the vector layer on the map
p = vector.process()
m.clear()
m.addLayer(p.toLayer())
m.zoomToImageExtent(p)

inter.identifyPopup(m,p)

# Display the trivariate choropleth legend
display(HTML(svg))
```

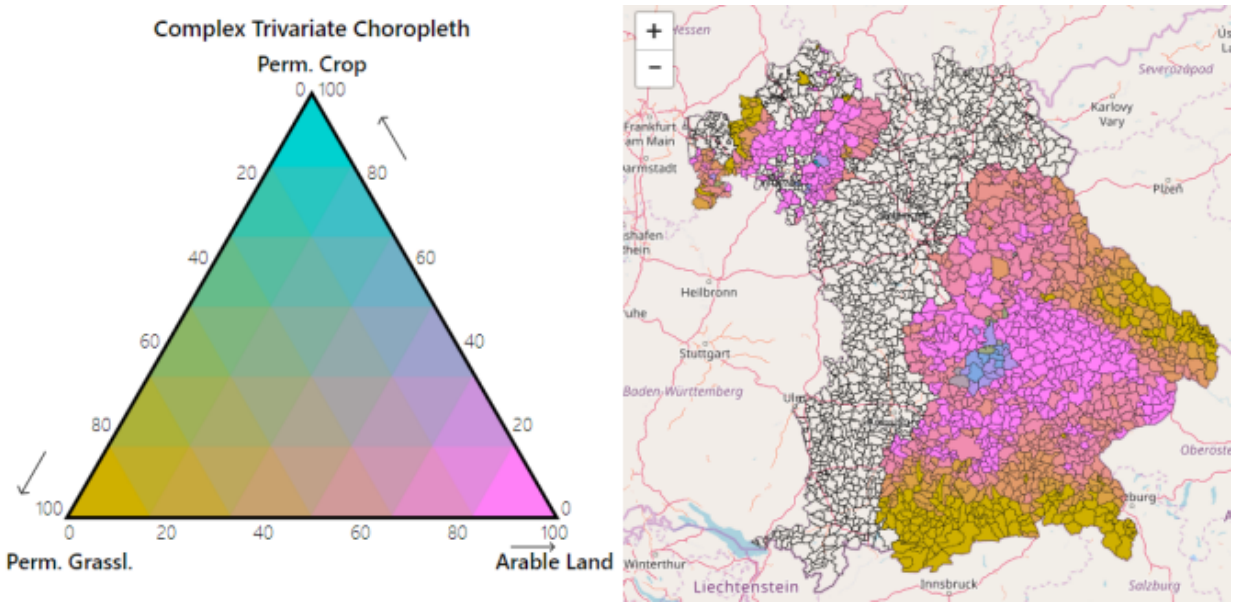


Fig. 3.12: Example of an interactive map showing polygons colored with a complex trivariate choropleth legend.

3.3.1.6 ipytrees module

Utility functions for the creation ipytrees from hierarchical data.

class ipytrees.**DataNode**(***kwargs: Any*)

ipytrees.**createIpytreeFromDF2Columns**(*df*, *colindexLabels1*, *colindexLabels2*, *colindexValues=-1*, *rootName=''*, *handle_click=<function basic_handle_click>*, *select_root=False*)

Create a two level ipytree from two columns of a Pandas DataFrame

Parameters

- **df** (*Pandas DataFrame*) – Input Pandas DataFrame

- **colindexLabels1** (*int*) – Index of the column that contains the labels of the first level of the tree
- **colindexLabels2** (*int*) – Index of the column that contains the labels of the second level of the tree
- **colindexValues** (*int, optional*) – Index of the column that contains the values for the nodes of the tree
- **rootName** (*str, optional*) – Name to be displayed as root of the tree (default is '')
- **handle_click** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking (default is ipytree.basic_handle_click)
- **select_root** (*bool, optional*) – If True the root node is selected at start (default is False)

Returns
A tuple of 3 elements

Return type
the tree instance, a dict containing the info on the nodes, a dict containing the parent of each of the nodes

ipytrees.**createIpytreeFromList**(*nameslist=[], rootName='', separator='.', valuefor={}, handle_click=<function basic_handle_click>, select_root=False*)

Create a ipytree from a list of names with an implicit tree structure, example: ['JRC', 'JRC.D', 'JRC.D.3', ...]

Parameters

- **nameslist** (*list of strings, optional*) – List of strings that contain a hierarchical structure, considering the separator character (default is [])
- **rootName** (*str, optional*) – Name to be displayed as root of the tree (default is '')
- **separator** (*str, optional*) – String or character to be considered as separator for extracting the hierarchical structure from the nameslist list of strings (default is '.')
- **valuefor** (*dict, optional*) – Dictionary to assign a numerical value to each node of the tree (default is {})
- **handle_click** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking (default is ipytree.basic_handle_click)
- **select_root** (*bool, optional*) – If True the root node is selected at start (default is False)

Returns
A tuple of 3 elements

Return type
the tree instance, a dict containing the info on the nodes, a dict containing the parent of each of the nodes

Example

Example of the creation of a ipytree from a list of strings with hierarchy defined by the '.' character:

```
from vois import ipytrees

def onclick(event):
    if event['new']:
        print(event.owner, event.owner.value)

tree, n, p = ipytrees.createIpytreeFromList(['A','A.1','A.2',
                                             'A.1.1','A.3.1',
                                             'A.4.1.2','A.5.2.1',
                                             'A.4.2.3.1','B','B.A'],
                                             rootName='Directorates',
                                             valuefor={'A.1.1': 10.0,
                                                       'A.3.1': 5.0},
                                             handle_click=onclick)
display(tree)
```

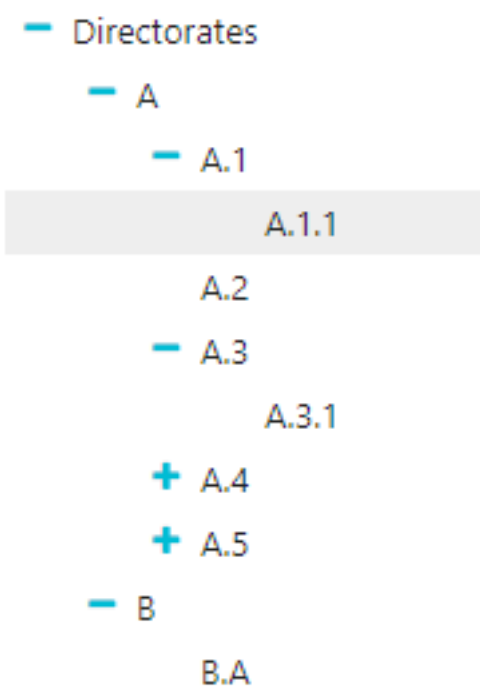


Fig. 3.13: Ipytree produced by the example code

3.3.1.7 leafletMap module

Utility functions for the creation of interactive maps using ipyleaflet Map.

`leafletMap.countriesMap(df, code_column=None, value_column='value', codes_selected=[], center=None, zoom=None, width='99%', height='400px', min_width=None, basemap={'attribution': '(C) OpenStreetMap contributors', 'html_attribution': '© OpenStreetMap contributors', 'max_zoom': 19, 'name': 'OpenStreetMap.Mapnik', 'url': 'https://tile.openstreetmap.org/{z}/{x}/{y}.png'}, detailedcountries=False, colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921'], stdevnumber=2.0, stroke='#232323', stroke_selected='#00ffff', stroke_width=3.0, decimals=2, minallowed_value=None, maxallowed_value=None, style={'dashArray': '0', 'fillOpacity': 0.6, 'opacity': 1}, hover_style={'dashArray': '0', 'fillOpacity': 0.85, 'opacity': 1})`

Creation of an interactive map to display the countries of the world. An input Pandas DataFrame df is used to join a column of numeric values to the countries, using the iso2code (ISO 3166-2) as internal key attribute. Once the values are assigned to the countries, a graduated legend is calculated based on mean and standard deviation of the assigned values. A input list of colors is used to represent the countries given their assigned value.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to the countries. It has to contain at least a column with numeric values.
- **code_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the unique code of the countries in the ISO-3166-2 standard. This column is used to perform the join with the internal attribute of the countries vector dataset that contains the country code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the countries using the join on the ISO-3166-2 codes (default is 'value')
- **codes_selected** (*list of strings, optional*) – List of codes of countries to display as selected (default is [])
- **center** (*tuple of (lat,lon), optional*) – Geographical coordinates of the initial center of the interactive map visualization (default is None)
- **zoom** (*int, optional*) – Initial zoom level of the interactive map (default is None)
- **width** (*str, optional*) – Width of the map widget to create (default is '99%')
- **height** (*str, optional*) – Height of the map widget to create (default is '400px')
- **min_width** (*str, optional*) – Minimum width of the layout of the map widget (default is None)
- **basemap** (*instance of basemaps type, optional*) – Basemap to use as background map (default is basemaps.OpenStreetMap.Mapnik)
- **detailedcountries** (*bool, optional*) – If True loads the more detailed version of the countries dataset (default is False()
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)

- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to country polygons and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **stroke** (*str, optional*) – Color to use for the border of countries (default is '#232323')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected countries (default is '#00ffff')
- **stroke_width** (*float, optional*) – Width of the border of the country polygons in pixels (default is 3.0)
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **style** (*dict, optional*) – Style to apply to the features (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.6})
- **hover_style** (*dict, optional*) – Style to apply to the features when hover (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.85})

Return type
a ipyleaflet.Map instance

Example

Creation of a map displaying a random variable on 4 european countries. The numerical values assigned to each of the countries are randomly generated using numpy.random.uniform and saved into a dictionary having the country code as the key. This dict is transformed to a Pandas DataFrame with 4 rows and having 'iso2code' and 'value' as columns. The graduated legend is build using the 'inverted' Reds Plotly colorscale (low values are dark red, intermediate values are red, high values are white):

```
import numpy as np
import pandas as pd
import plotly.express as px
from ipyleaflet import basemaps
from vois import leafletMap

countries = ['DE', 'ES', 'FR', 'IT']

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

m = leafletMap.countriesMap(df,
```

(continues on next page)

(continued from previous page)

```
                                code_column='iso2code',
                                height='400px',
                                stroke_width=2.0,
                                stroke_selected='yellow',
                                basemap=basemaps.Stamen.Terrain,
                                colorlist=px.colors.sequential.Reds[::-1],
                                codes_selected=['IT'],
                                center=[43,12], zoom=5)
display(m)
```

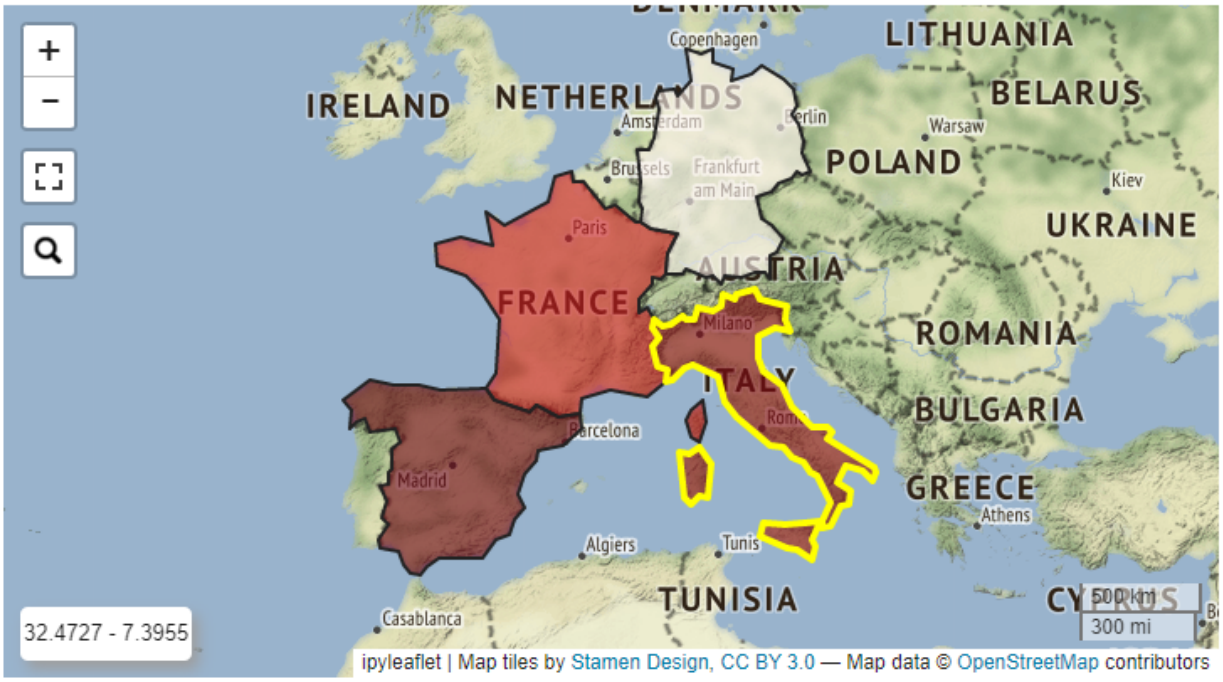


Fig. 3.14: Example of an ipyleaflet Map displaying 4 european countries.

`leafletMap.`**`geojsonCategoricalMap`**(*geojson_path, geojson_attribute, center=None, zoom=None, width='99%', height='400px', min_width=None, basemap={'attribution': '(C) OpenStreetMap contributors', 'html_attribution': '© OpenStreetMap contributors', 'max_zoom': 19, 'name': 'OpenStreetMap.Mapnik', 'url': 'https://tile.openstreetmap.org/{z}/{x}/{y}.png'}, colormap={}, stroke='#232323', stroke_width=3.0, fill='#aaaaaa', style={'dashArray': '0', 'fillOpacity': 0.6, 'opacity': 1}, hover_style={'dashArray': '0', 'fillOpacity': 0.85, 'opacity': 1}*)

Creation of an interactive map to display a custom geojson dataset where colors are assigned to feature based on the values of an internal attribute of the input geojson file. The colormap parameter is a dictionary with keys corresponding to all the unique values of the internal attribute, which are mapped to the colors to use for representing each class.

Parameters

- **geojson_path** (*str*) – Path of the geojson file to load that contains the geographic features in geojson format

- **geojson_attribute** (*str*) – Name of the attribute of the geojson dataset that contains the thematisatio attribute of the features. This attribute will be used to retrieve the colors to assign to the features using the colormap parameter
- **center** (*tuple of (lat,lon), optional*) – Geographical coordinates of the initial center of the interactive map visualization (default is None)
- **zoom** (*int, optional*) – Initial zoom level of the interactive map (default is None)
- **width** (*str, optional*) – Width of the map widget to create (default is '99%')
- **height** (*str, optional*) – Height of the map widget to create (default is '400px')
- **min_width** (*str, optional*) – Minimum width of the layout of the map widget (default is None)
- **basemap** (*basemap instance, optional*) – Basemap to use as background in the map visualization (default is basemaps.OpenStreetMap.Mapnik). See Documentation of ipyleaflet for details
- **colormap** (*dictionary containing geojson_attribute values as keys and colors as values*) – Colors to assign to each distinct value of the geojson_attribute
- **stroke** (*str, optional*) – Color to use for the border of polygons (default is '#232323')
- **stroke_width** (*float, optional*) – Width of the border of the polygons in pixels (default is 3.0)
- **fill** (*str, optional*) – Default fill color to use for the polygons (default is '#aaaaaa')
- **style** (*dict, optional*) – Style to apply to the features (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.6})
- **hover_style** (*dict, optional*) – Style to apply to the features when hover (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.85})

Return type
a ipyleaflet.Map instance

Example

Creation of a map displaying a custom geojson with data on landuse. The colors are assigned to the polygons based on the values of the 'fclass' attribute of the input geojson file. The colormap parameter is a dictionary with keys corresponding to all the unique landuse classes, which are mapped to the colors of a Plotly discrete color scale (see Color Sequences in Plotly Express):

```
import plotly.express as px
from IPython.display import display
from ipywidgets import widgets, Layout
from vois import leafletMap, svgUtils, geojsonUtils

# Load landuse example and get unique landuse classes
filepath = './data/landuse.geojson'
geojson = geojsonUtils.geojsonLoadFile(filepath)
landuses = sorted(list(set(geojsonUtils.geojsonAll(geojson,'fclass'))))

# Create a colormap (dictionary that maps landuses to colors)
colors   = px.colors.qualitative.Dark24
colormap = dict(zip(landuses, colors))
```

(continues on next page)

(continued from previous page)

```
m = leafletMap.geojsonCategoricalMap(filepath,
                                     'fclass',
                                     stroke_width=1.0,
                                     stroke='black',
                                     colormap=colormap,
                                     width='79%',
                                     height='700px',
                                     center=[51.005,13.6],
                                     zoom=12,
                                     basemap=basemaps.CartoDB.Positron,
                                     style={'opacity': 1, 'dashArray': '0',
↪'fillOpacity': 1})

outlegend = widgets.Output(layout=Layout(width='230px',height='680px'))
with outlegend:
    display(HTML(svgUtils.categoriesLegend("Landuse legend",
                                           landuses,
                                           colorlist=colors[:len(landuses)])))
widgets.HBox([m,outlegend])
```

leafletMap.**geojsonMap**(*df*, *geojson_path*, *geojson_attribute*, *code_column=None*, *value_column='value'*, *codes_selected=[]*, *center=None*, *zoom=None*, *width='99%'*, *height='400px'*, *min_width=None*, *basemap={'attribution': '(C) OpenStreetMap contributors', 'html_attribution': '© OpenStreetMap contributors', 'max_zoom': 19, 'name': 'OpenStreetMap.Mapnik', 'url': 'https://tile.openstreetmap.org/{z}/{x}/{y}.png'}*, *colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921']*, *stdevnumber=2.0*, *stroke='#232323'*, *stroke_selected='#00ffff'*, *stroke_width=3.0*, *decimals=2*, *minallowed_value=None*, *maxallowed_value=None*, *style={'dashArray': '0', 'fillOpacity': 0.6, 'opacity': 1}*, *hover_style={'dashArray': '0', 'fillOpacity': 0.85, 'opacity': 1}*)

Creation of an interactive map to display a custom geojson dataset. An input Pandas DataFrame df is used to join a column of numeric values to the geojson features, using the <geojson_attribute> as the internal key attribute. Once the values are assigned to the features, a graduated legend is calculated based on mean and standard deviation of the assigned values. A input list of colors is used to represent the featuress given their assigned value.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to features. It has to contain at least a column with numeric values.
- **geojson_path** (*str*) – Path of the geojson file to load that contains the geographic features in geojson format
- **geojson_attribute** (*str*) – Name of the attribute of the geojson dataset that contains the unique codes of the features. This attribute will be use as internal key in the join operation with the df Pandas DataFrame
- **code_column** (*str, optional*) – Name of the column of the df Pandas DataFrame containing the unique code of the features. This column is used to perform the join with the internal attribute of the geojson vector dataset that contains the unique code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)

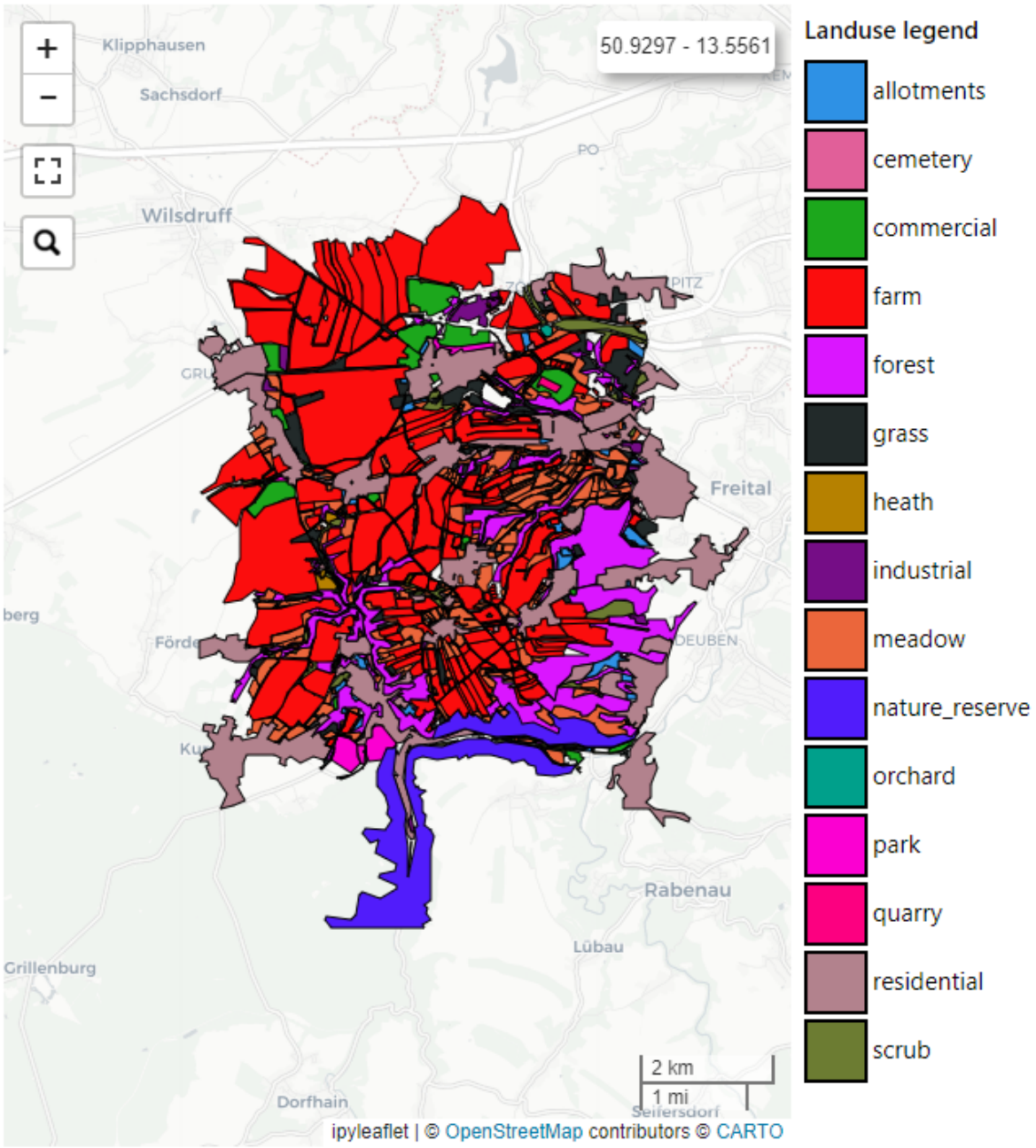


Fig. 3.15: Example of an ipyleaflet Map displaying 4 european countries from a custom geojson file.

- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the features using the join on geojson unique codes (default is 'value')
- **codes_selected** (*list of strings, optional*) – List of codes of features to display as selected (default is [])
- **center** (*tuple of (lat,lon), optional*) – Geographical coordinates of the initial center of the interactive map visualization (default is None)
- **zoom** (*int, optional*) – Initial zoom level of the interactive map (default is None)
- **width** (*str, optional*) – Width of the map widget to create (default is '99%')
- **height** (*str, optional*) – Height of the map widget to create (default is '400px')
- **min_width** (*str, optional*) – Minimum width of the layout of the map widget (default is None)
- **basemap** (*basemap instance, optional*) – Basemap to use as background in the map visualization (default is basemaps.OpenStreetMap.Mapnik). See Documentation of ipyleaflet for details
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)
- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to features and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **stroke** (*str, optional*) – Color to use for the border of polygons (default is '#232323')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected polygons (default is '#00ffff')
- **stroke_width** (*float, optional*) – Width of the border of the polygons in pixels (default is 3.0)
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **style** (*dict, optional*) – Style to apply to the features (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.6})
- **hover_style** (*dict, optional*) – Style to apply to the features when hover (default is {'opacity': 1, 'dashArray': '0', 'fillOpacity': 0.85})

Return type

a ipyleaflet.Map instance

Example

Creation of a map displaying a custom geojson. The numerical values assigned to each of the countries are randomly generated using numpy.random.uniform and saved into a dictionary having the country code as the key. This dict is transformed to a Pandas DataFrame with 4 rows and having 'iso2code' and 'value' as columns. The graduated legend is build using the 'inverted' Viridis Plotly colorscale:

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import leafletMap

countries = ['DE', 'ES', 'FR', 'IT']

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

m = leafletMap.geojsonMap(df,
                          './data/ne_50m_admin_0_countries.geojson',
                          'ISO_A2_EH',   # Internal attribute used as key
                          code_column='iso2code',
                          height='400px',
                          stroke_width=1.5,
                          stroke_selected='yellow',
                          colorlist=px.colors.sequential.Viridis[::-1],
                          codes_selected=['IT'],
                          center=[43,12], zoom=5)
display(m)
```

3.3.1.8 svgBarChart module

SVG BarChart to display interactive vertical bars.

svgBarChart.**svgBarChart**(*title=''*, *width=30.0*, *height=40.0*, *names=[]*, *values=[]*, *stddevs=None*, *dictnames=None*, *selectedname=None*, *fontsize=1.1*, *titlecolor='black'*, *barstrokecolor='black'*, *xaxistextcolor='black'*, *xaxistextsizemultiplier=1.0*, *xaxistextangle=0.0*, *xaxistextextraspace=0.0*, *yaxistextextraspace=5.0*, *xaxistextdisplacey=0.0*, *valuestextsizemultiplier=0.7*, *valuestextangle=0.0*, *strokew_axis=0.2*, *strokew_horizontal_lines=0.06*, *strokecol_axis='#bbbbbb'*, *strokecol_horizontal_lines='#dddddd'*, *showvalues=False*, *textweight=400*, *colorlist=['rgb(247,251,255)', 'rgb(222,235,247)', 'rgb(198,219,239)', 'rgb(158,202,225)', 'rgb(107,174,214)', 'rgb(66,146,198)', 'rgb(33,113,181)', 'rgb(8,81,156)', 'rgb(8,48,107)']*, *colors_on_minmax_values=True*, *fixedcolors=False*, *enabledeselect=False*, *selectcolor='red'*, *showselection=False*, *hovercolor='yellow'*, *valuedigits=4*, *barpercentwidth=90.0*, *stdevnumber=2.0*, *minallowed_value=None*, *maxallowed_value=None*, *yaxis_min=None*, *yaxis_max=None*, *on_change=None*)

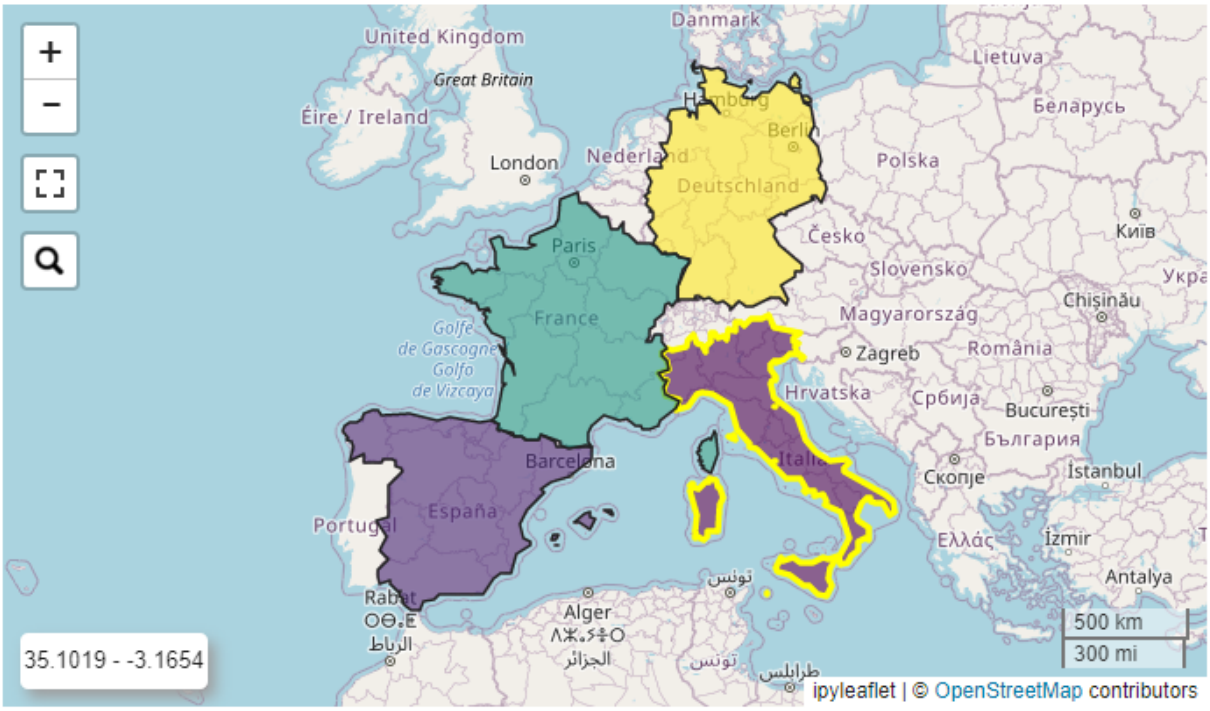


Fig. 3.16: Example of an ipyleaflet Map displaying 4 european countries from a custom geojson file.

Creation of a vertical bar chart given a list of labels and corresponding numerical values. Click on the rectangles is managed by calling a custom python function.

Parameters

- **title** (*str, optional*) – Title of the chart (default is 'Ranking of labels')
- **width** (*float, optional*) – Width of the chart in vw units (default is 20.0)
- **height** (*float, optional*) – Height of the chart in vh units (default is 90.0)
- **names** (*list of str, optional*) – List of names to display inside the rectangles (default is [])
- **values** (*list of float, optional*) – List of numerical values of the same length of the names list (default is [])
- **stddevs** (*list of float, optional*) – List of numerical values representing the standard deviation of the values, to be displayed on top of the columns (default is None)
- **dictnames** (*dict, optional*) – Dictionary to convert codes to names when displaying the selection (default is None)
- **selectedname** (*str, optional*) – Name of the selected item (default is None)
- **fontsize** (*float, optional*) – Size of the standard font to use for names in vh coordinates (default is 1.1vh). The chart title will be displayed with sizes proportional to the fontsize parameter (up to two times for the chart title)
- **titlecolor** (*str, optional*) – Color to use for the chart title (default is 'black')
- **barstrokecolor** (*str, optional*) – Color for the bars border (default is 'black')

- **xaxistextcolor** (*str, optional*) – Color of labels on the X axis (default is 'black')
- **xaxistextsizemultiplier** (*float, optional*) – Multiplier factor to calculate the x axis label size from the default fontsize (default is 1.0)
- **xaxistextangle** (*float, optional*) – Angle in degree to rotate x axis labels (default is 0.0)
- **xaxistextextraspace** (*float, optional*) – Extra space to reserve to xaxis labels (default is 0.0)
- **yaxistextextraspace** (*float, optional*) – Extra space to reserve to yaxis labels in percentage (default is 5.0)
- **xaxistextdisplacey** (*float, optional*) – Positional displace in y coordinate to apply to the xaxis labels (default is 0.0)
- **valuestextsizemultiplier** (*float, optional*) – Multiplier factor to calculate the values text size from the default fontsize (default is 0.7)
- **valuestextangle** (*float, optional*) – Angle in degree to rotate the values text on top of the bars (default is 0.0)
- **strokew_axis** (*float, optional*) – Stroke width of the lines that define the x and y axis (default is 0.2)
- **strokew_horizontal_lines** (*float, optional*) – Stroke width of the secondary horizontal lines starting from the y axis (default is 0.06)
- **strokecol_axis** (*str, optional*) – Color to use for the lines of the x and y axis (default is "#bbbbbb")
- **strokecol_horizontal_lines** (*str, optional*) – Color to use for the secondary horizontal lines starting from the y axis (default is "#dddddd")
- **showvalues** (*bool, optional*) – If True the value of each bar is shown on top of the bar (default is False)
- **textweight** (*int, optional*) – Weight of the text written inside the rectangles (default is 400). The chart title will be displayed with weight equal to textweight+100
- **colorlist** (*list of colors, optional*) – List of colors to assign to the rectangles based on the numerical values (default is the inverted Plotly px.colors.sequential.Blues, see Plotly sequential color scales and Plotly qualitative color sequences)
- **fixedcolors** (*bool, optional*) – If True, the list of colors is assigned to the values in their original order (and colorlist must contain the same number of elements!). Default is False
- **colors_on_minmax_values** (*bool, optional*) – If True, the colors are stretched on the min and max effective values, otherwise on the minallowed,maxallowed values range (default is True)
- **enabledeselect** (*bool, optional*) – If True, a click on a selected element deselects it, and the on_change function is called with None as argument (default is False)
- **selectcolor** (*str, optional*) – Color to use for the border of the selected rectangle (default is 'red')
- **showselection** (*bool, optional*) – If True, the currently selected bar is framed with the selection color (default is False)
- **hovercolor** (*str, optional*) – Color to use for the border on hovering the rectangles (default is 'yellow')

- **valuedigits** (*int, optional*) – Number of digits to use for the display of the values (default is 4)
- **barpercentwidth** (*float, optional*) – Percentage of element width occupied by the bar. The remaining percentage of the element width is the space before the next element. Default is 90.0.
- **stdevnumber** (*float, optional*) – The correspondance between the values and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **yaxis_min** (*float, optional*) – Minimum value displayed on the y axis (default is None)
- **yaxis_max** (*float, optional*) – Maximum value displayed on the y axis (default is None)
- **on_change** (*function, optional*) – Python function to call when the selection of the rectangle items changes (default is None). The function is called with a tuple as unique argument. The tuple will contain (name, value, originalposition) of the selected rectangle

Return type

an instance of widgets.Output with the svg chart displayed in it

Example

Creation of a SVG chart to display a vertical bar chart:

```
from IPython.display import display
from ipywidgets import widgets
import numpy as np
import plotly.express as px
from vois import eucountries as eu
from vois import svgBarChart

# Names of EU countries
names  = [c.iso2code for c in eu.countries.EuropeanUnion()]

# Randomly generated values for each country
values = np.random.uniform(low=0.1, high=1.0, size=(len(names)))

# Randomly generated stdevs for each country
stddevs = np.random.uniform(low=0.01, high=0.2, size=(len(names)))

debug = widgets.Output()
display(debug)

def on_change(arg):
    with debug:
        print(arg)
```

(continues on next page)

(continued from previous page)

```
out = svgBarChart.svgBarChart(title='Sample Bar Chart',
                              names=names,
                              values=values,
                              stddevs=stddevs,
                              width=39.0,
                              height=35.0,
                              fontsize=0.7,
                              barstrokecolor='#44444400',
                              xaxistextcolor='#666666',
                              showvalues=True,
                              colorlist=px.colors.sequential.Viridis,
                              hovercolor='blue',
                              stdevnumber=100.0,
                              valuedigits=2,
                              barpercentwidth=90.0,
                              enabledeselect=True,
                              showselection=False,
                              minallowed_value=0.0,
                              on_change=on_change)

display(out)
```

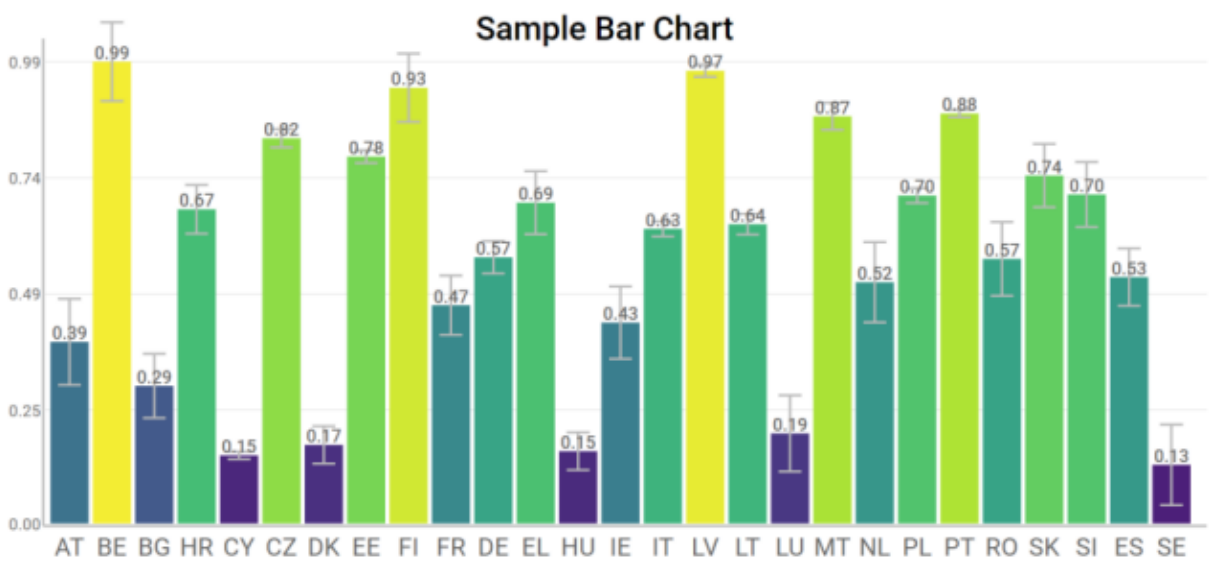


Fig. 3.17: Example of an interactive vertical bars chart

3.3.1.9 svgBubblesChart module

SVG bubbles chart from a pandas DataFrame.

class svgBubblesChart.**svgBubblesChart**(*df*, *xcolumn*, *ycolumn*, *sizecolumn*, *colorcolumn*, *width=100.0*, *height=50.0*, *xstart=6.0*, *strokewidth=1*, *strokecolor='#ffffff'*, *backcolor='#aaaaaa'*, *backlinecolor='#888888'*, *bubblecolors=['rgb(27,158,119)', 'rgb(217,95,2)', 'rgb(117,112,179)', 'rgb(231,41,138)', 'rgb(102,166,30)', 'rgb(230,171,2)', 'rgb(166,118,29)', 'rgb(102,102,102)']*, *textcolor='black'*, *textweight=400*, *fontsize=1.1*, *xtextangle=0.0*, *title=''*, *mode='spread'*, *legendrows=2*, *legenditemwidth=10*)

Creation of a bubbles chart given an input DataFrame. It is a convenient chart for representing a numerical value that depends on 3 discrete variables. It displays a bi-dimensional grid where the unique values of the x column are displayed on the X axis, while the unique values of the y column are displayed on the Y axis. Inside each cell of the grid, a group of bubbles is displayed, one for each distinct value of the color column, while the size of the circles is proportional to the numerical value read from the size column. See the below example on a mushrooms dataset (taken from https://www.kaggle.com/datasets/uciml/mushroom-classification). The SVG chart has the x coordinates expressed in vw coordinates and the y coordinates expressed in vh coordinates. Cliks on the legend is managed so that individual color categories can be excluded from the chart.

Parameters

- **df** (*pandas DataFrame*) – Input DataFrame
- **xcolumn** (*str*) – Name of the DataFrame column containing the values to be displayed on the X axis
- **ycolumn** (*str*) – Name of the DataFrame column containing the values to be displayed on the X axis
- **sizecolumn** (*str*) – Name of the DataFrame column containing the numerical values to be used to size the bubbles
- **colorcolumn** (*str*) – Name of the DataFrame column containing the values to be used to give a color to the bubbles and to display the color legend
- **width** (*float, optional*) – Width of the chart in vw units (default is 100.0)
- **height** (*float, optional*) – Height of the chart in vh units (default is 50.0)
- **xstart** (*float, optional*) – X coordinate on vw units where the grid starts (default is 6.0). This value can be used to leave more or less space to the Y axis strings
- **strokewidth** (*int, optional*) – Width in pixels of the stroke to use for the bubbles (default is 1)
- **strokecolor** (*str, optional*) – Color of the stroke to use for the bubbles (default is '#ffffff')
- **backcolor** (*str, optional*) – Background color of the grid (default is '#aaaaaa')
- **backlinecolor** (*str, optional*) – Color of the lines defining the cells of the grid (default id '#888888')
- **bubblecolors** (*list of colors, optional*) – List of colors to use for the bubble. Each color is assigned to one of the unique values of the color column of the input DataFrame (default is px.colors.qualitative.Dark2)
- **textcolor** (*str, optional*) – Color to use for the texts, chart title, axis titles, axis values, etc. (default is 'black')

- **textweight** (*int, optional*) – Weight of the text (default is 400). The chart title, the axis titles and the legend title will be displayed with weight equal to textweight+100
- **fontsize** (*float, optional*) – Size of the standard font to use for values displayed in the X and Y axis in vh coordinates (default is 1.1vh). The chart title, the axis titles and the legend title will be displayed with sizes proportional to the fontsize parameter (up to two times for the chart title)
- **xtextangle** (*float, optional*) – Angle to use for the rotation of values displayed on the X axis (default is 0.0)
- **title** (*str, optional*) – Title of the chart (default is '')
- **mode** (*str, optional*) – Mode to use for the placement of bubbles inside the cells of the grid. Possible values are 'spread' (circles are horizontally displaced), 'concentric' (all circles have their centre in the center of the cell) or 'tangent' (circles are all tangent on the center-bottom point of the cell). Default is 'spread'.
- **legendrows** (*int, optional*) – Number of rows to use for the legend (default is 2)
- **legenditemwidth** (*int, optional*) – Width in percent of the total width of each item of the legend (default is 10)

Return type
an instance of widgets.Output with the svg chart displayed in it

Example

Creation of a SVG bubble chart chart to display some of the columns of the mushroom dataset (see https://www.kaggle.com/datasets/uciml/mushroom-classification):

```
from IPython.display import display
import pandas as pd
import plotly.express as px
colorlist = px.colors.qualitative.Dark2
from vois import svgBubblesChart

df = pd.read_csv('https://jeodpp.jrc.ec.europa.eu/services/shared/csv/mushrooms.csv
↪')

xcolumn     = 'cap-shape'
ycolumn     = 'cap-surface'
colorcolumn = 'habitat'
sizecolumn  = 'count'
dfgrouped = df.groupby([xcolumn, ycolumn, colorcolumn]).size().reset_
↪index(name=sizecolumn)

dfgrouped.columns = ['Shape', 'Cap surface', 'Mushroom habitat:', 'count']

b = svgBubblesChart.svgBubblesChart(dfgrouped,
                                    height=50.0,
                                    xcolumn=dfgrouped.columns[0],
                                    ycolumn=dfgrouped.columns[1],
                                    colorcolumn=dfgrouped.columns[2],
                                    sizecolumn=dfgrouped.columns[3],
                                    strokewidth=1,
```

(continues on next page)

(continued from previous page)

```
                                    strokecolor='#ffffff',
                                    backcolor='#f0f0f0',
                                    backlinecolor='#000000',
                                    bubblecolors=colorlist,
                                    fontsize=1.1,
                                    title='Mushrooms analysis',
                                    mode='spread')
display(b.draw())
display(HTML(b.getlegendsvg()))
```

getlegendsvg()

Returns a string containing the SVG legend for the sizes of the circles.

3.3.1.10 svgGraph module

SVG visualization of a graph.

svgGraph.**svgGraph**(*nodes_name*, *nodes_label*, *nodes_color*, *nodes_pos*, *edges*, *edge_label='Value'*, *textcolor='white'*, *edgestrokecolor='lightgrey'*, *edgestrokewidth=0.3*, *nodestrokecolor='grey'*, *nodestrokewidth=0.3*, *selectedcolor='red'*, *selectedstrokewidth=0.7*, *width=300.0*, *heightpercent=100.0*, *borderspercent=10.0*, *nodesradius=3.0*, *fontsize=3.0*, *onclick=None*)

Display of a graph.

Parameters

- **nodes_name** (*dict*) – Dictionary with Key=nodeid and Value=name (short label to display inside the node)
- **nodes_label** (*dict*) – Dictionary with Key=nodeid and Value=description (long description to display as tooltip of the nodes)
- **nodes_color** (*dict*) – Dictionary with Key=nodeid and Value=color to use for the node
- **nodes_pos** (*dict*) – Dictionary with Key=nodeid and Value=[x,y] coordinates of the node
- **edges** (*dict*) – Dictionary with Key=(nodeid1,nodeid2) and Value containing a numerical value to display in the tooltip
- **edge_label** (*str, optional*) – String to display inside the tooltip over an edge of the graph (default is 'Value')
- **textcolor** (*str, optional*) – Color to use for text of nodes (default is 'white')
- **edgestrokecolor** (*str, optional*) – Color to use to display the edges (default is 'lightgrey')
- **edgestrokewidth** (*float, optional*) – Stroke width of the line used to display the edges (default is 0.3)
- **nodestrokecolor** (*str, optional*) – Color to use for the border of the nodes (default is 'grey')
- **nodestrokewidth** (*float, optional*) – Stroke width of the line used to display the border of the nodes (default is 0.3)
- **selectedcolor** (*str, optional*) – Color to use to display the border of the selected node (default is 'red')

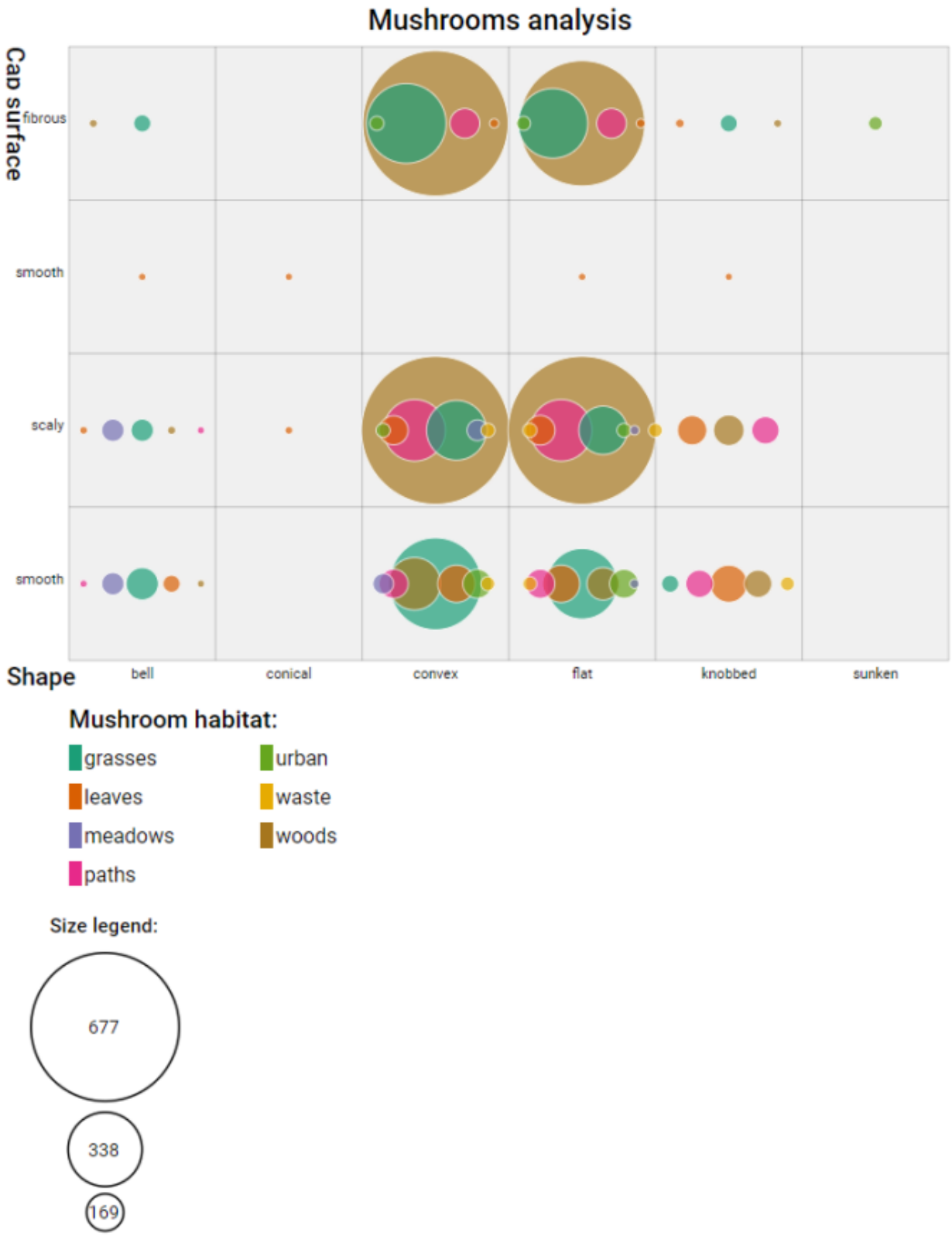


Fig. 3.18: Example of an interactive bubbles chart in SVG

- **selectedstrokewidth** (*float, optional*) – Stroke width of the line used to display the border of the selected node (default is 0.7)
- **width** (*int or float or str, optional*) – Width of the drawing. If an integer or a float is passed, the size is intended in pixels units, otherwise a string containing the units must be passed (example: '4vw'). The default is 300.0 for 300 pixels
- **heightpercent** (*float, optional*) – Height of the drawing in percentage of the Width (default is 100.0, meaning that a square chart will be created)
- **borderspercent** (*float, optional*) – Border to add in percentage on each side of the chart to be sure that the nodes circles are completely inside (default is 10.0)
- **nodesradius** (*float, optional*) – Radius of the circles that represent the nodes. The total graph drawing is created in SVG coordinates [0,100]. Default is 3.0.
- **fontsize** (*float, optional*) – Size of the font used for nodes' texts. The total graph drawing is created in SVG coordinates [0,100]. Default is 3.0.
- **onclick** (*function, optional*) – Python function to call when the user clicks on one of the nodes of the graph. The function will receive as parameter the nodeid of the clicked node, or -1 when the click is outside of all the nodes

Return type

an ipywidgets.Output instance with the graph displayed inside

Example

Example of a generation and display of a random graph using networkx library and svgGraph module:

```
from vois import svgGraph
import networkx as nx
import random
r = lambda: random.randint(0,255)

# Generate a random graph using networkx
G = nx.gnp_random_graph(20, 0.2, seed=12345)

# Assign position to nodes using Fruchterman-Reingold force-directed algorithm
nodes_pos = nx.spring_layout(G, weight='weight', seed=12345)

# Extract nodes and edges information from the graph
nodes = list(G.nodes)
edges = list(G.edges)

# Generate dictionaries requested by the svgGraph module
nodes_name  = dict(zip(nodes,['N'+str(x) for x in nodes]))
nodes_label = dict(zip(nodes,['Description of node '+str(x) for x in nodes]))
nodes_color = dict(zip(nodes,['#%02X%02X%02X' % (r(),r(),r()) for x in nodes]))  #␣
↪Random colors

edges_value = dict(zip(edges,[random.randint(0,100) for x in edges]))

# Display the graph
svgGraph.svgGraph(nodes_name, nodes_label, nodes_color, nodes_pos, edges_value,␣
↪width=800)
```

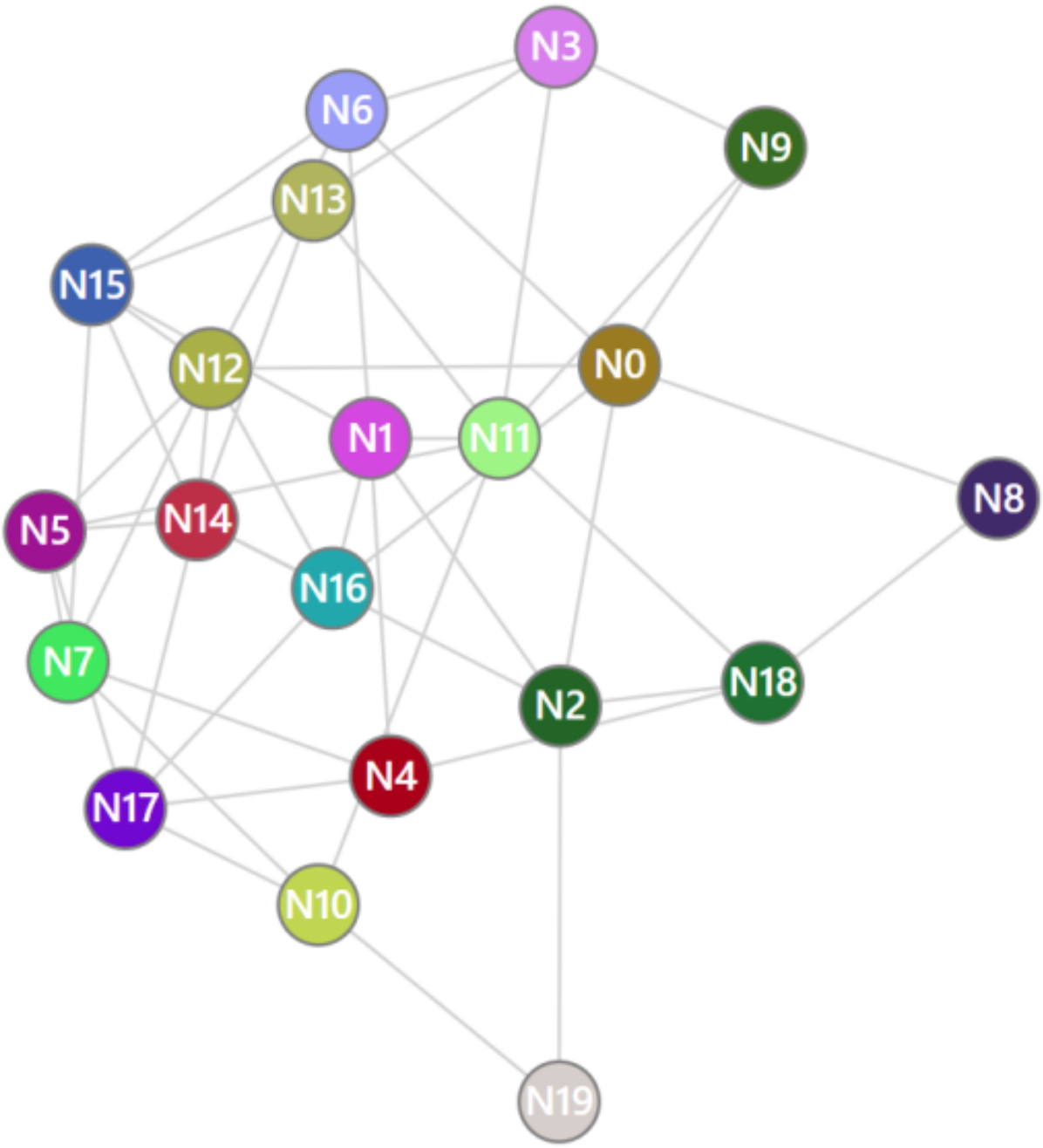


Fig. 3.19: Example of a graph visualization

3.3.1.11 svgHeatmap module

SVG heatmap chart from a pandas DataFrame.

`svgHeatmap.heatmapChart(df, width=100.0, height=50.0, hTitle=5.0, wTitle=15.0, columnTitleMaxChar=40, textRows='Row', textColumns='Column', textValues='Value', title='Heatmap', fontsize=1.0, colorlist=['rgb(247,251,255)', 'rgb(222,235,247)', 'rgb(198,219,239)', 'rgb(158,202,225)', 'rgb(107,174,214)', 'rgb(66,146,198)', 'rgb(33,113,181)', 'rgb(8,81,156)', 'rgb(8,48,107)'], textcolor='black', textweight=400, backcolor='white', highlitecolor='#426bb4', highliteback='#dddddd', minvalue=0.0, maxvalue=1.0, decimals=2)`

Creation of a heatmap chart given an input DataFrame containing only numbers. The index strings and column names are taken as names for rows and columns. The SVG chart has x coordinates expressed in vw coordinates and the y coordinates expressed invh coordinates. Rows and columns of the chart can be selected and the chart will be sorted on decreasing values (when a column is selected, the rows are sorted, and viceversa)

Parameters

- **df** (*pandas DataFrame*) – Input DataFrame containing only numbers
- **width** (*float, optional*) – Width of the chart in vw units (default is 100.0)
- **height** (*float, optional*) – Height of the chart in vh units (default is 50.0)
- **hTitle** (*float, optional*) – Height of the Title bar in percentage on the height of the chart (default is 5.0)
- **wTitle** (*float, optional*) – Width of the left space for row titles in percentage of the width pf the chart (default is 15.0)
- **columnTitleMaxChar** (*int, optional*) – Max number of chars for the column names (longer names are splitted), (default is 40)
- **textRows** (*str, optional*) – Title for the rows (default is 'Row')
- **textColumns** (*str, optional*) – Title for the columns (default is 'Column')
- **textValues** (*str, optional*) – Text to label the values in the tooltip when the mouse is over a cell of the chart (default is 'Value')
- **title** (*str, optional*) – Title of the chart (default is 'Heatmap')
- **fontsize** (*float, optional*) – Size of the standard font to use for values displayed in the X and Y axis in vh coordinates (default is 1.0vh). The chart title and the axis titles will be displayed with sizes proportional to the fontsize parameter (up to two times for the chart title)
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Blues, see Plotly sequential color scales and Plotly qualitative color sequences)
- **textcolor** (*str, optional*) – Color to use for rows and columns text (default is 'black')
- **textweight** (*int, optional*) – Weight of the text (default is 400). The chart title and the axis titles will be displayed with weight equal to textweight+100
- **backcolor** (*str, optional*) – Background color (default is 'white')
- **highlitecolor** (*str, optional*) – Color to use for displaying text of the selected row and colum (default is '#426bb4')

- **highliteback** (*str, optional*) – Color to use as background of the selected row and column (default is '#dddddd')
- **minvalue** (*float, optional*) – Minimum value of the DataFrame cells to be used for color assignment (default is 0.0)
- **maxvalue** (*float, optional*) – Minimum value of the DataFrame cells to be used for color assignment (default is 1.0)
- **decimals** (*int, optional*) – Number of decimals for the tooltip display of cell values (default is 2)

Return type
an instance of widgets.Output with the svg chart displayed in it

Example

Creation of a SVG heatmap chart to display a matrix of random numbers:

```
from IPython.display import display
import pandas as pd
import numpy as np
from vois import svgHeatmap

df = pd.DataFrame(np.random.random((20,50)))
display(svgHeatmap.heatmapChart(df, wTitle=7.0, hTitle=70)
```

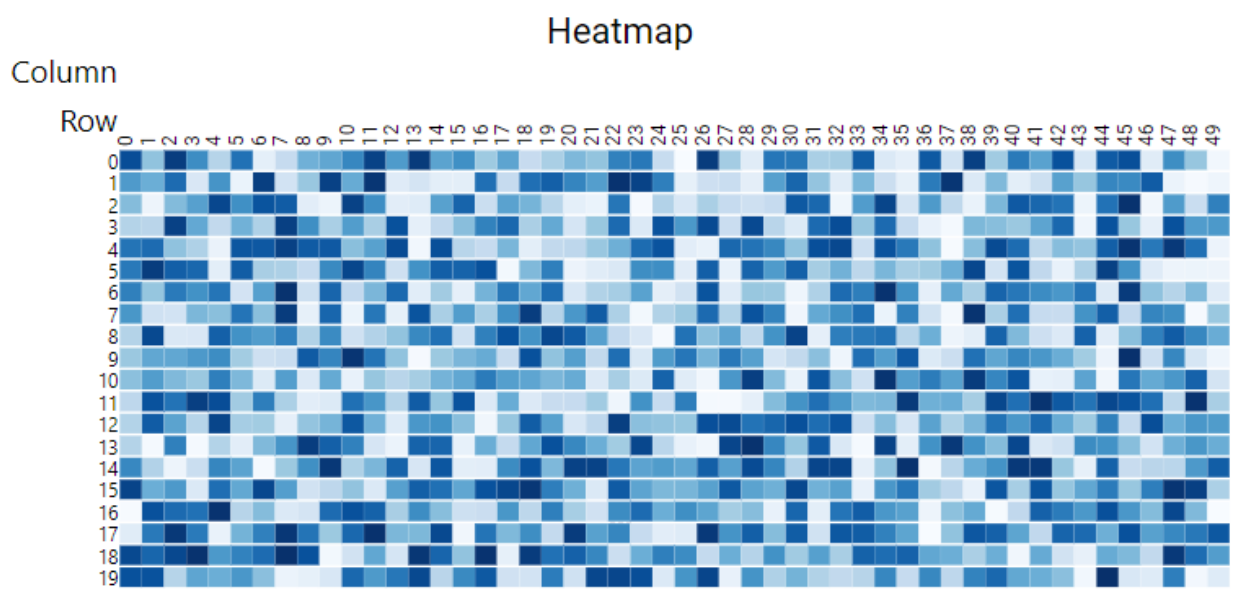


Fig. 3.20: Example of an interactive heatmap chart in SVG

3.3.1.12 svgMap module

European map implemented in SVG.

`svgMap.country_codes = ['LV', 'AL', 'CH', 'AT', 'HR', 'ES', 'IS', 'RO', 'IT', 'VA', 'HU', 'SE', 'NL', 'CZ', 'DE', 'FR', 'ME', 'BE', 'SI', 'LU', 'IE', 'BA', 'MC', 'BG', 'PL', 'LI', 'GB', 'RS', 'SM', 'DK', 'IM', 'EE', 'SK', 'EL', 'LT', 'NO', 'PT', 'AD', 'MK', 'MT', 'GI', 'FI', 'XK', 'CY']`

List of all the codes for the countries managed by the svgMap module, following the EUROSTAT country codes.

See EUROSTAT country codes

Type
List of 2 characters strings

`svgMap.country_name = {'AD': 'Andorra', 'AL': 'Albania', 'AT': 'Austria', 'BA': 'Bosnia and Herzegovina', 'BE': 'Belgium', 'BG': 'Bulgaria', 'CH': 'Switzerland', 'CY': 'Cyprus', 'CZ': 'Czechia', 'DE': 'Germany', 'DK': 'Denmark', 'EE': 'Estonia', 'EL': 'Greece', 'ES': 'Spain', 'FI': 'Finland', 'FR': 'France', 'GB': 'United Kingdom', 'GI': 'Gibraltar', 'HR': 'Croatia', 'HU': 'Hungary', 'IE': 'Ireland', 'IM': 'Isle of Man', 'IS': 'Iceland', 'IT': 'Italy', 'LI': 'Liechtenstein', 'LT': 'Lithuania', 'LU': 'Luxembourg', 'LV': 'Latvia', 'MC': 'Monaco', 'ME': 'Montenegro', 'MK': 'North Macedonia', 'MT': 'Malta', 'NL': 'Netherlands', 'NO': 'Norway', 'PL': 'Poland', 'PT': 'Portugal', 'RO': 'Romania', 'RS': 'Republic of Serbia', 'SE': 'Sweden', 'SI': 'Slovenia', 'SK': 'Slovakia', 'SM': 'San Marino', 'VA': 'Vatican', 'XK': 'Kosovo'}`

dict with key=EUROSTAT code of countries, value=name of the country.

`svgMap.svgMapEurope(df, code_column=None, value_column='value', label_column='label', codes_selected=[], width=400, colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921'], stdevnumber=2.0, fill='#f1f1f1', stroke='#232323', stroke_selected='#00ffff', stroke_width=3.0, onhoverfill='yellow', decimals=2, minallowed_value=None, maxallowed_value=None, hoveronempty=False, legendtitle='', legendunits='', bordercolor='black', textcolor='black', dark=False)`

Static map of European countries with color legend obtained by joining with a Pandas DataFrame.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to the countries. It has to contain at least a column with numeric values.
- **code_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the unique code of the countries in the EUROSTAT Country Codes standard (see https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes). This column is used to perform the join with the internal attribute of the countries vector dataset that contains the country code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the countries using the join on the EUROSTAT Country Codes (default is 'value')
- **codes_selected** (*list of strings, optional*) – List of codes of countries to display as selected (default is [])
- **width** (*int, optional*) – Width of the map in pixels (default is 400)
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)

- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to country polygons and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **fill** (*str, optional*) – Fill color to use for the countries that are not joined (default is '#f1f1f1')
- **stroke** (*str, optional*) – Color to use for the border of countries (default is '#232323')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected countries (default is '#00ffff')
- **stroke_width** (*float, optional*) – Width of the border of the country polygons in pixels (default is 3.0)
- **onhoverfill** (*str, optional*) – Color for highlighting countries on hover (default is 'yellow')
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **hoveronempty** (*bool, optional*) – If True highlights polygon on hover even if no value present in input df for the polygon (default is False)
- **legendtitle** (*str, optional*) – Title to add on top of the legend (default is '')
- **legendunits** (*str, optional*) – Units of measure to add to the bottom of the legend (default is '')
- **bordercolor** (*str, optional*) – Color for lines and rects of the legend (default is 'black')
- **textcolor** (*str, optional*) – Color for texts of the legend (default is 'black')
- **dark** (*bool, optional*) – If True, the bordercolor and textcolor are set to white (default is False)

Return type

a string containing SVG text to display the map of European countries

Example

Map of European countries joined with random values in [0,100]:

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import svgMap

countries = svgMap.country_codes

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
```

(continues on next page)

(continued from previous page)

```
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

svg = svgMap.svgMapEurope(df,
                          code_column='iso2code',
                          width=650,
                          stdevnumber=1.5,
                          colorlist=px.colors.sequential.Viridis,
                          stroke_width=4.0,
                          stroke_selected='red',
                          onhoverfill='#f8bd1a',
                          codes_selected=['IT'],
                          legendtitle='Legent title',
                          legendunits='KTOE per 100K inhabit.')
display(HTML(svg))
```

3.3.1.13 svgPackedCirclesChart module

SVG Packed Circles chart from a pandas DataFrame.

`svgPackedCirclesChart.`**`svgPackedCirclesChart`**(*df*, *valuecolumn*, *labelcolumn*, *dimension=30.0*, *colorlist=['rgb(247,251,255)', 'rgb(222,235,247)', 'rgb(198,219,239)', 'rgb(158,202,225)', 'rgb(107,174,214)', 'rgb(66,146,198)', 'rgb(33,113,181)', 'rgb(8,81,156)', 'rgb(8,48,107)']*, *title=''*, *titlecolor='black'*, *titleweight=600*, *titlecentered=False*, *titlesize=1.6*, *labelcolor='black'*, *fontsize=1.3*, *drawscale=True*, *scaledigits=2*)

Creation of a packed circles chart given an input DataFrame. Labels are taken from the labelcolumn column of the DataFrame, and numerical values from the valuecolumn column. The chart displays the values with proportional size circles packed toward the centre of the chart.

Parameters

- **df** (*pandas DataFrame*) – Input DataFrame
- **valuecolumn** (*str*) – Name of the DataFrame column containing the values to be used for the size of the circles
- **labelcolumn** (*str*) – Name of the DataFrame column containing the description of each circle
- **dimension** (*float, optional*) – Side of the square containing the chart in vw coordinates (default is 20.0). If drawscale is True, the total height of the chart will be titleh+dimension+fontsize+0.5 vw, otherwise it will be
- **colorlist** (*list of colors, optional*) – List of colors to use for the circles, given their size (default is px.colors.sequential.Blues)
- **title** (*str, optional*) – Title of the chart (default is ‘’)

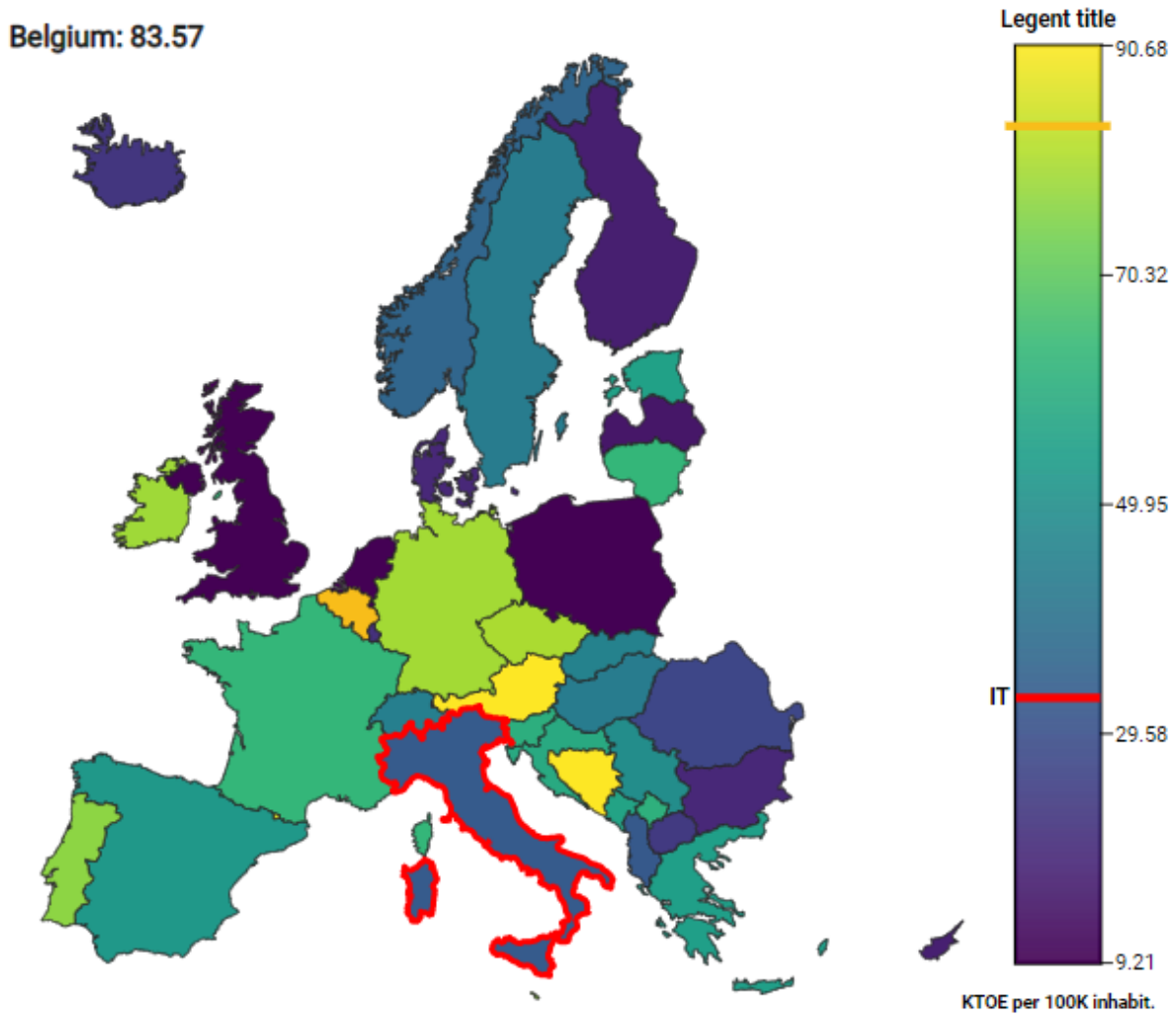


Fig. 3.21: Example of a static map of European countries created as an SVG drawing

- **titlecolor** (*str, optional*) – Color to use for the title (default is 'black')
- **titleweight** (*int, optional*) – Font weight to use for the title (default is 600)
- **titlecentered** (*bool, optional*) – If True the title will be displayed in center-top position (default is False)
- **titlesize** (*float, optional*) – Font size in vw coordinate to use for the title of the chart (default is 1.3)
- **labelcolor** (*str, optional*) – Color to use for the labels of the circles (default is 'black')
- **fontsize** (*float, optional*) – Font size in vw coordinates to use for the labels of the circles (default is 1.0)
- **drawscale** (*bool, optional*) – If Trues, the chart will display an horizontal scale below the circles (default is True)
- **scaledigits** (*int, optional*) – Number of decimal digits to display in the tooltip of the scalebar (default is 2)

Return type
a string containing SVG code that can be displayed using display(HTML(...))

Example

Creation of a SVG packed circles chart chart to display some values relater to years:

```
from IPython.display import display
import pandas as pd
import plotly.express as px
from vois import svgPackedCirclesChart

table = [['Year', 'Occurrencies'],
         [2016, 251],
         [2017, 239],
         [2018, 186],
         [2019, 142],
         [2020, 59],
         [2021, 47],
         [2022, 95],
]

headers = table.pop(0)
df = pd.DataFrame(table, columns=headers)

svg = svgPackedCirclesChart.svgPackedCirclesChart(df,
                                                  'Occurrencies',
                                                  'Year',
                                                  dimension=310,
                                                  colorlist=px.colors.sequential.
↪YlOrRd,
                                                  labelcolor='#aaaaaa',
                                                  fontsize=13,
                                                  title='Occurrencies per year:')
display(HTML(svg))
```

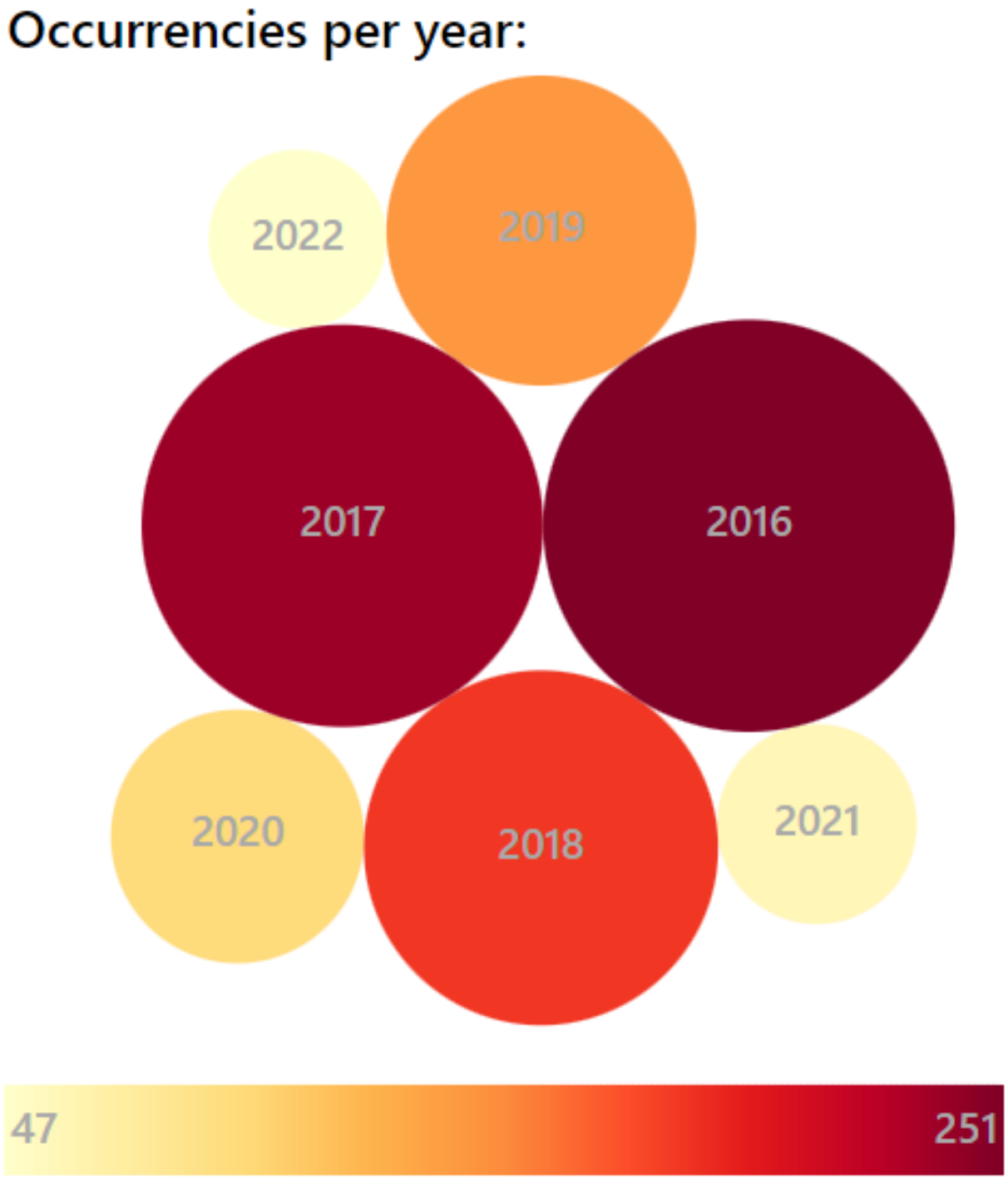


Fig. 3.22: Example of a packed circles chart in SVG

3.3.1.14 svgRankChart module

SVG RankChart to display vertically aligned rectangles.

`svgRankChart.svgRankChart(title='Ranking of labels', width=20.0, height=90.0, names=[], values=[], splitnamelenght=40, addposition=True, fontsize=1.1, titlecolor='black', textweight=400, colorlist=['rgb(247,251,255)', 'rgb(222,235,247)', 'rgb(198,219,239)', 'rgb(158,202,225)', 'rgb(107,174,214)', 'rgb(66,146,198)', 'rgb(33,113,181)', 'rgb(8,81,156)', 'rgb(8,48,107)'], fixedcolors=False, enabledeselect=False, selectfirstatstart=True, selectcolor='red', hovercolor='yellow', stdevnumber=2.0, minallowed_value=None, maxallowed_value=None, on_change=None)`

Creation of a chart given a list of labels and corresponding numerical values. The labels are ordered according to the decreasing values and displayed as a vertical list of rectangles. Click on the rectangles is managed by calling a custom python function.

Parameters

- **title** (*str, optional*) – Title of the chart (default is 'Ranking of labels')
- **width** (*float, optional*) – Width of the chart in vw units (default is 20.0)
- **height** (*float, optional*) – Height of the chart in vh units (default is 90.0)
- **names** (*list of str, optional*) – List of names to display inside the rectangles (default is [])
- **values** (*list of float, optional*) – List of numerical values of the same length of the names list (default is [])
- **splitnamelenght** (*int, optional*) – Maximum number of characters to display in a row. If the name length is higher, it is splitted in two rows (default is 40)
- **addposition** (*bool, optional*) – If True, the position is added in front of the name (starting from 1 and determined by the accompanying value). Default is True
- **fontsize** (*float, optional*) – Size of the standard font to use for names in vh coordinates (default is 1.1vh). The chart title will be displayed with sizes proportional to the fontsize parameter (up to two times for the chart title)
- **titlecolor** (*str, optional*) – Color to use for the chart title (default is 'black')
- **textweight** (*int, optional*) – Weight of the text written inside the rectangles (default is 400). The chart title will be displayed with weight equal to textweight+100
- **colorlist** (*list of colors, optional*) – List of colors to assign to the rectangles based on the numerical values (default is the inverted Plotly px.colors.sequential.Blues, see Plotly sequential color scales and Plotly qualitative color sequences)
- **fixedcolors** (*bool, optional*) – If True, the list of colors is assigned to the values in their original order (and colorlist must contain the same number of elements!). Default is False
- **enabledeselect** (*bool, optional*) – If True, a click on a selected element deselects it, and the on_change function is called with None as argument (default is False)
- **selectfirstatstart** (*bool, optional*) – If True, at start the rectangle corresponding to the greatest value is selected (default is True)
- **selectcolor** (*str, optional*) – Color to use for the border of the selected rectangle (default is 'red')

- **hovercolor** (*str, optional*) – Color to use for the border on hovering the rectangles (default is 'yellow')
- **stdevnumber** (*float, optional*) – The correspondance between the values and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **on_change** (*function, optional*) – Python function to call when the selection of the rectangle items changes (default is None). The function is called with a tuple as unique argument. The tuple will contain (name, value, originalposition) of the selected rectangle

Return type

an instance of widgets.Output with the svg chart displayed in it

Example

Creation of a SVG chart to display some names ordered by correspondent values:

```
from IPython.display import display
from ipywidgets import widgets
from vois import svgRankChart
import numpy as np
import plotly.express as px

# List of sentences
names = ['European society needs to grasp the opportunities brought by the digital␣
↪transformation',
         'A deep transformation such as the one facilitated by digital technologies
↪',
         'The progress needs to be evenly distributed across all regions',
         'Over the next decade the EU economy and society need to undergo a␣
↪profound transformation',
         'The design of green and digital policy actions needs to consider socio-
↪economic and territorial impacts',
         'The EU supports the shift to a sustainable and resilient growth model',
         'Address the challenges arising from the demographic transition',
         'Geospatial data and methods are (usually) globally applicable',
         'Considerable EU investments',
         'The acceleration of the implementation of the Fit for 55 package',
         'To become climate-neutral by 2050, Europe needs to decarbonise',
         'Efforts need to be intensified in the harder-to-decarbonise sectors',
         'The Commission recently decided to better understand the environment␣
↪interface',
         'One Health was already recognised by the Commission as an emerging␣
↪priority',
         'Achieving sustainability requires a holistic, well-coordinated approach',
         'The conflict in Ukraine is endangering food security and has implications␣
```

(continues on next page)

(continued from previous page)

```
↪for food supply chains',
         'The food system is composed of sub-systems and interacts with other key␣
↪systems',
         'The future of the EU and its position in the world will be influenced by␣
↪population trajectories',
         'Policymakers are often asked to react quickly to new circumstances',
         'Obtaining economic benefit from natural resources whilst maintaining␣
↪natural capital',
         'We must be able to create EU policies that  decouple resource use from␣
↪economic development',
         'Robust, resilient and innovative EU economy is a necessary condition for␣
↪ensuring the well-being'
         ]

# Randomly generated values for each sentence
values = np.random.uniform(low=0.5, high=25.0, size=(len(names,)))

debug = widgets.Output()
display(debug)

def on_change(arg):
    with debug:
        print(arg)

out = svgRankChart.svgRankChart(names=names,
                                values=values,
                                width=20.0,
                                height=90.0,
                                splitnamelenght=45,
                                addposition=False,
                                fontsize=1.3,
                                selectfirstatstart=False,
                                colorlist=px.colors.sequential.Viridis,
                                hovercolor='blue',
                                on_change=on_change)
display(out)
```

3.3.1.15 svgUtils module

SVG drawings for general use.

svgUtils.**AnimatedPieChart**(*values=[10.0, 25.0, 34.0, 24.0, 23.0], colors=['#2d82c2', '#95cb92', '#e7ee99', '#ffde88', '#ff945a', '#e34e4f'], labels=None, duration=0.75, fillpercentage=2, backcolor='#f1f1f1', dimension=400, textcolor='black', fontsize=15, textweight=400, decimals=1, centertext='', centercolor='black', centerfontsize=18, centertextweight=500, onclick=None, additional_argument=None, is_selected=False, displayvalues=True*)

Creation of an animated pie chart in SVG format. Given an array of float values, and optional labels, the function draws a pie chart that fills its slices with a short animation. An ipywidgets.Output instance is returned, which has the SVG chart displayed in it. By passing a value to the onclick parameter, it is possible to manage the click

Fig. 3.23: Example of an interactive and ordered list of rectangles

event on the slices of the pie, providing interactivity to the drawing. The capture of the click event is done using the ipyevents library .

Parameters

- **values** (*list of float values, optional*) – List of float values that represent the relative dimension of each of the slices (default is [10.0, 25.0, 34.0, 24.0, 23.0])
- **colors** (*list of strings representing colors, optional*) – Colors to use for each of the slices of the pie (default is ['#2d82c2', '#95cb92', '#e7ee99', '#ffde88', '#ff945a', '#e34e4f'])
- **labels** – Labels for each of the slices of the pie (default is None)
- **strings** (*list of*) – Labels for each of the slices of the pie (default is None)
- **optional** – Labels for each of the slices of the pie (default is None)
- **duration** (*float, optional*) – Duration in seconds of the animation (default is 0.75 seconds)
- **fillpercentage** (*int, optional (in 0,1,2,3,4)*) – Amount of the circle that is filled by the slices (default is 2 which means 60%)
- **backcolor** (*str, optional*) – Background color of the pie (default is '#f1f1f1')
- **dimension** (*int or float or str, optional*) – Side of the drawing. If an integer or a float is passed, the size is intended in pixels units, otherwise a string containing the units must be passed (example: '4vh'). Th default is 400.0 for 400 pixels
- **textcolor** (*str, optional*) – Color of text
- **fontsize** (*int, optional*) – Dimension of text in pixels (default is 15)
- **textweight** (*int, optional*) – Weight of text (default is 400, >= 500 is Bold)
- **decimals** (*int, optional*) – Number of decimal to use for the display of numbers (default is 1)
- **centertext** (*str, optional*) – Text string to display at the center of the pie (default is '')
- **centercolor** (*str, optional*) – Color to use for the central text
- **centerfontsize** (*int, optional*) – Text dimension for the text displayed at the center of the pie
- **centertextweight** (*int, optional*) – Weight of central text (default is 500, >= 500 is Bold)
- **onclick** (*function, optional*) – Python function to call when the user clicks on one of the slices of the pie. The function will receive as first parameter the index of the clicked slice, and the additional_argument as second parameter
- **additional_argument** (*any, optional*) – Additional parameter passed to the onclick function when the user clicks on one of the slices of the pie (default is None)
- **is_selected** (*bool, optional*) – Flag to select the pie chart (default is False)
- **displayvalues** (*bool, optional*) – If True each slide of the pie will display, inside parenthesis, the corresponding value (default is True)

Return type

a tuple containing an instance of ipywidgets.Output() widget, and a string containing the SVG code of the drawing

Example

Example of a pie chart:

```
from vois import svgUtils
import plotly.express as px
from ipywidgets import widgets

debug = widgets.Output()
display(debug)

def onclick(arg):
    with debug:
        print('clicked %s' % arg)

out, txt = svgUtils.AnimatedPieChart(values=[10.0, 25.0, 18.0, 20.0, 9.5],
                                     labels=['Option<br>1', 'Option<br>2',
                                             'Option 3', 'Option 4',
                                             'Others'],
                                     centerfontsize=28,
                                     fontsize=16, textweight=400,
                                     colors=px.colors.qualitative.D3,
                                     backcolor='#dfdfdf',
                                     centertext='Example Pie',
                                     onclick=onclick,
                                     dimension=380.0,
                                     duration=1.0)
display(out)
```

`svgUtils.SmallCircle(text1, text2, percentage, forecolor='#308040', backcolor=None, textcolor='white', dimension=300.0, fontsize=16.0, textsize=0.0)`

Display of a circle graphics displaying a text and a percentage value. It shows an animation to reach the requested percentage of the full circle.

Parameters

- **text1** (*str*) – First string of text to display inside the circle (usually a brief description of the variable displayed)
- **text2** (*str*) – Second string of text to display inside the circle (usually the numercal value)
- **percentage** (*float*) – Value in percentage to be displayed
- **forecolor** (*str, optional*) – Color to use for the circle border (default is '#308040')
- **backcolor** (*str, optional*) – Color to use for the interior of the circle (default is None, so a lighter color is generated from the forecolor)
- **textcolor** (*str, optional*) – Color to use for text (default is 'white')
- **dimension** (*int or float or str, optional*) – Side of the drawing. If an integer or a float is passed, the size is intended in pixels units, otherwise a string containing the units must be passed (example: '4vh'). The default is 300.0 for 300 pixels
- **textsize** (*float, optional*) – Text dimension in pixels (default is 0.0 which means that it is automatically calculated from the dimension of the drawing)

Return type

an ipywidgets.Output instance with the circle already displayed inside

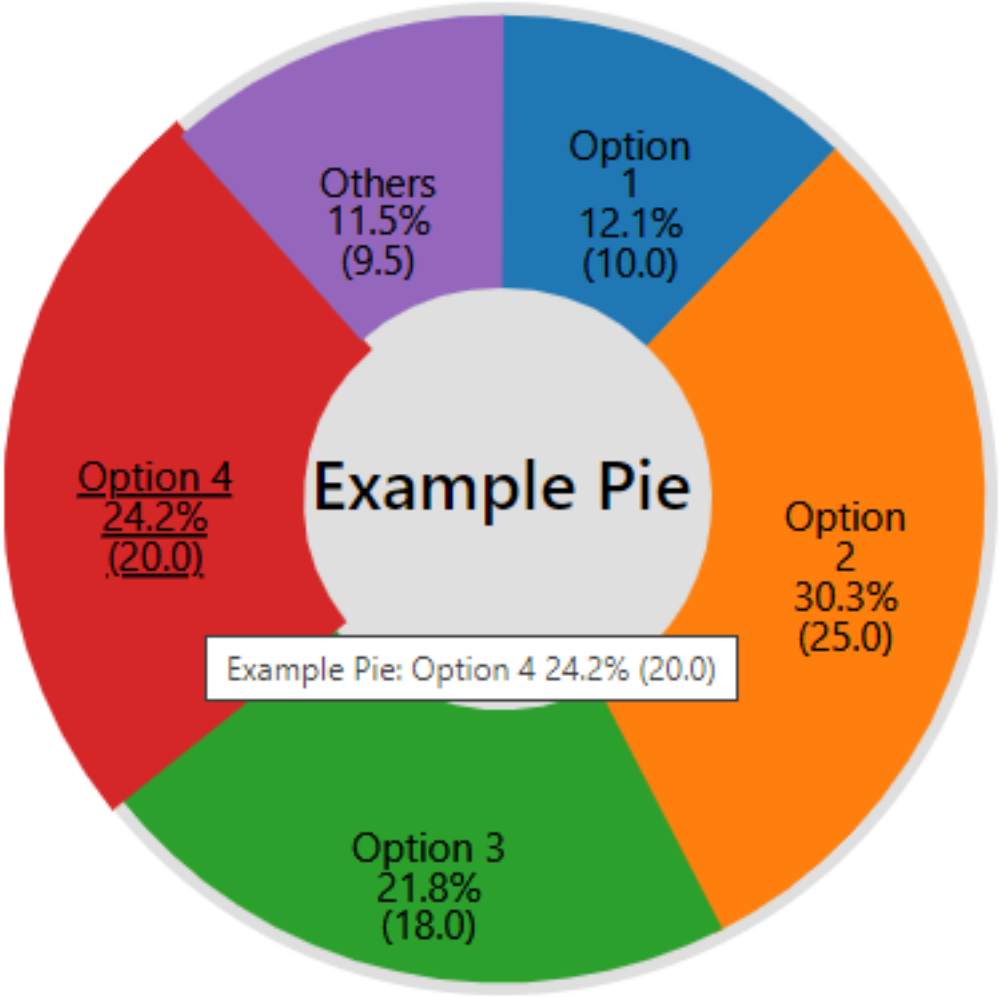


Fig. 3.24: Example of an animated SVG to graphically represent a pie chart

Example

Example of a circle to represent a percentage with an animation:

```
from vois import svgUtils
from random import randrange

percentage = randrange(1000)/10.0
svgUtils.SmallCircle('Green<br>deal',
                     '%.1f%%' % percentage,
                     percentage,
                     forecolor="#308040",
                     dimension=200)
```

Fig. 3.25: Example of an animated SVG to graphically represent a percentage value

`svgUtils.categoriesLegend`(*title*, *descriptions*, *width=200*, *elemHeight=38*, *bordercolor='black'*, *borderWidth=2*, *textcolor='black'*, *colorlist=['#fef4ef', '#fdd3c0', '#fca080', '#fa6a49', '#e23026', '#b11117', '#66000c']*, *dark=False*)

Creation of a legend for categories given title and descriptions of the classes.

Parameters

- **title** (*str*) – Title of the legend
- **descriptions** (*list of strings*) – Description of each of the classe of the legend. If a description contains a ' ' character, the text after it is displayed as multi-line text in smaller font
- **width** (*int, optional*) – Width in pixel of the legend (default is 200)
- **elemHeight** (*int, optional*) – Height in pixels of each of the elements of the legend (default is 38)
- **bordercolor** (*str, optional*) – Color of the border of the legend elements (default is 'black')
- **borderWidth** (*int, optional*) – Width in pixels of the border of the legend elements (default is 2)

- **textcolor** (*str, optional*) – Color of text for the legend elements (default is 'black')
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (see Plotly sequential color scales and Plotly qualitative color sequences)
- **dark** (*bool, optional*) – If True, the bordercolor and textcolor are set to white (default is False)

Return type
a string containing SVG text to display the legend

Example

Example of the creation of an SVG drawing for a categories legend:

```
from vois import svgUtils
import plotly.express as px

svg = svgUtils.categoriesLegend("Legend title",
                                ['1:    Very long class description that can span␣
↪multiple lines and that contains no info at all',
                                 'Class 2', 'Class 3', 'Class 4'],
                                colorlist=px.colors.sequential.Blues,
                                width=250)
display(HTML(svg))
```

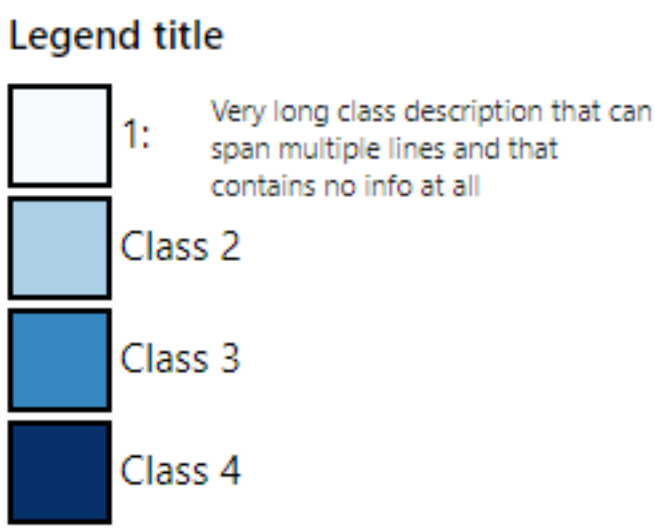


Fig. 3.26: Example of a categories legend

`svgUtils.graduatedLegend`(*df, code_column=None, value_column='value', label_column='label', dictnames=None, codes_selected=[], colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921'], stdevnumber=2.0, fill='#f1f1f1', stroke_selected='#00ffff', decimals=2, minallowed_value=None, maxallowed_value=None, hoveronempty=False, legendtitle='', legendunits='', fontsize=20, width=200, height=600, bordercolor='black', textcolor='black', dark=False*)

Creation of graduated legend in SVG format. Given a Pandas DataFrame in the same format of the one in input to `interMap.geojsonMap()` function, this functions generates an SVG drawing displaying a graduated colors legend.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to features. It has to contain at least a column with numeric values.
- **code_column** (*str, optional*) – Name of the column of the df Pandas DataFrame containing the unique code of the features. This column is used to perform the join with the internal attribute of the geojson vector dataset that contains the unique code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the features using the join on geojson unique codes (default is 'value')
- **label_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the label to be assigned to the features using the join on geojson unique codes (default is 'label')
- **dictnames** (*dict, optional*) – Dictionary to convert codes to names when displaying the selection (default is None)
- **codes_selected** (*list of strings, optional*) – List of codes of features to display as selected (default is [])
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)
- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to features and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **fill** (*str, optional*) – Fill color to use for the features that are not joined (default is '#f1f1f1')
- **stroke_selected** (*str, optional*) – Color to use for the selected features (default is '#00fff')
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **hoveronempty** (*bool, optional*) – If True highlights polygon on hover even if no value present in input df for the feature (default is False)
- **legendtitle** (*str, optional*) – Title to add on top of the legend (default is '')
- **legendunits** (*str, optional*) – Units of measure to add to the bottom of the legend (default is '')
- **fontsize** (*int, optional*) – Size in pixels of the font used for texts (default is 20)

- **width** (*int, optional*) – Width of the SVG drawing in pixels (default is 200)
- **height** (*int, optional*) – Height of the SVG drawing in pixels (default is 600)
- **bordercolor** (*str, optional*) – Color for lines and rects of the legend (default is 'black')
- **textcolor** (*str, optional*) – Color for texts of the legend (default is 'black')
- **dark** (*bool, optional*) – If True, the bordercolor and textcolor are set to white (default is False)

Return type
a string containing SVG text to display the graduated legend

Example

Creation of a SVG drawing to display a graduated legend. Input is prepared in the same way of the example provided for the *interMap.geojsonMap()* function:

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import svgMap, svgUtils

countries = svgMap.country_codes

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

svg = svgUtils.graduatedLegend(df, code_column='iso2code',
                               codes_selected=['IT', 'FR', 'CH'],
                               stroke_selected='red',
                               colorlist=px.colors.sequential.Viridis[::-1],
                               stdevnumber=2.0,
                               legendtitle='2020 Total energy consumption',
                               legendunits='KTOE per 100K inhabit.',
                               fontsize=18,
                               width=340, height=600)
display(HTML(svg))
```

svgUtils.**graduatedLegendVWVH**(*df, code_column=None, value_column='value', label_column='label', codes_selected=[], colorlist=['#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786', '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921'], stdevnumber=2.0, fill='#f1f1f1', stroke_selected='#00ffff', decimals=2, minallowed_value=None, maxallowed_value=None, hoveronempty=False, legendtitle='', legendunits='', fontsize=20, width=20.0, height=40.0, bordercolor='black', textcolor='black', dark=False*)

Creation of graduated legend in SVG format. Given a Pandas DataFrame in the same format of the one in input to *interMap.geojsonMap()* function, this functions generates an SVG drawing displaying a graduated colors legend.

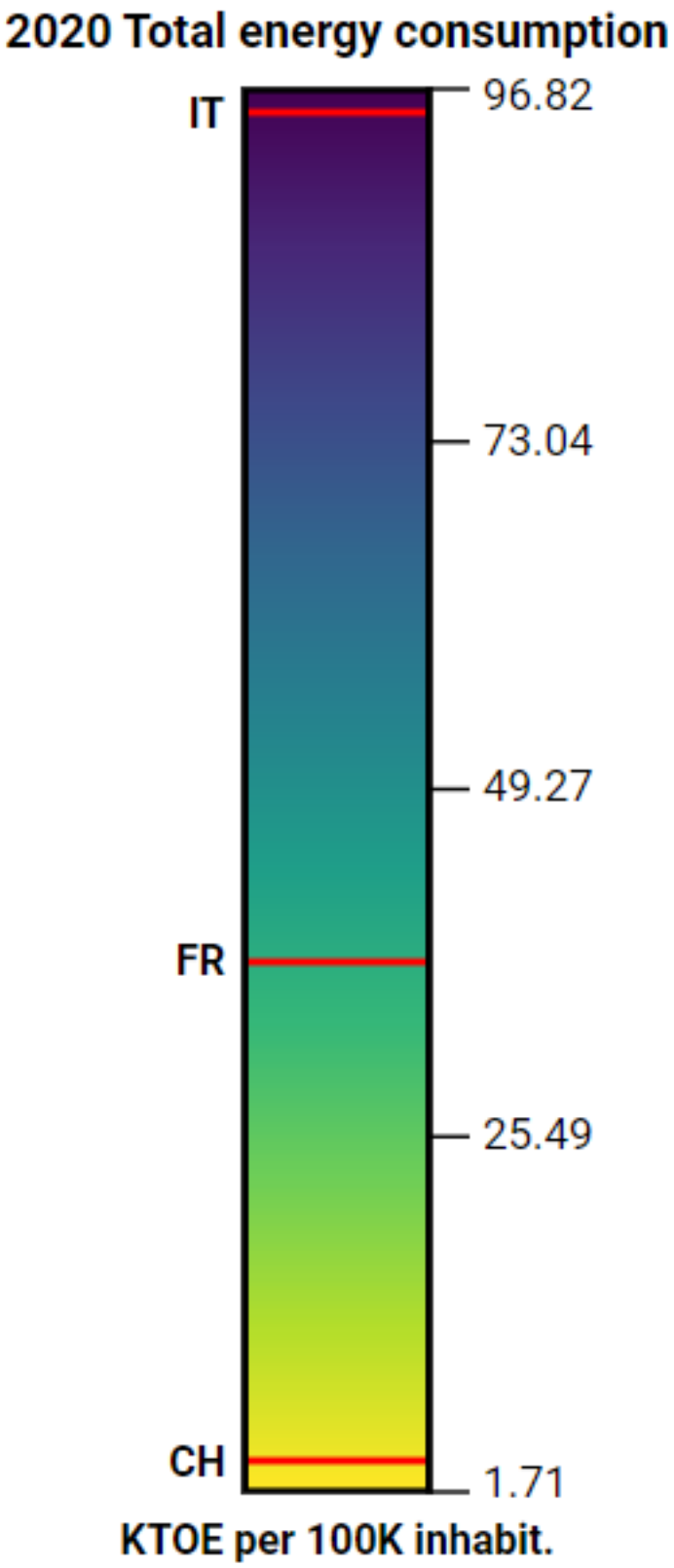


Fig. 3.27: Example of a graduatedLegend in SVG

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame to use for assigning values to features. It has to contain at least a column with numeric values.
- **code_column** (*str, optional*) – Name of the column of the df Pandas DataFrame containing the unique code of the features. This column is used to perform the join with the internal attribute of the geojson vector dataset that contains the unique code. If the code_column is None, the code is taken from the index of the DataFrame, (default is None)
- **value_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the values to be assigned to the features using the join on geojson unique codes (default is 'value')
- **label_column** (*str, optional*) – Name of the column of the Pandas DataFrame containing the label to be assigned to the features using the join on geojson unique codes (default is 'label')
- **codes_selected** (*list of strings, optional*) – List of codes of features to display as selected (default is [])
- **colorlist** (*list of colors, optional*) – List of colors to assign to the country polygons (default is the Plotly px.colors.sequential.Plasma, see Plotly sequential color scales and Plotly qualitative color sequences)
- **stdevnumber** (*float, optional*) – The correspondance between the values assigned to features and the colors list is done by calculating a range of values [min,max] to linearly map the values to the colors. This range is defined by calculating the mean and standard deviation of the country values and applying this formula [mean - stdevnumber*stddev, mean + stdevnumber*stddev]. Default is 2.0
- **fill** (*str, optional*) – Fill color to use for the features that are not joined (default is '#f1f1f1')
- **stroke_selected** (*str, optional*) – Color to use for the border of the selected features (default is '#00ffff')
- **decimals** (*int, optional*) – Number of decimals for the legend numbers display (default is 2)
- **minallowed_value** (*float, optional*) – Minimum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **maxallowed_value** (*float, optional*) – Maximum value allowed, to force the calculation of the [min,max] range to map the values to the colors
- **hoveronempty** (*bool, optional*) – If True highlights polygon on hover even if no value present in input df for the feature (default is False)
- **legendtitle** (*str, optional*) – Title to add on top of the legend (default is '')
- **legendunits** (*str, optional*) – Units of measure to add to the bottom of the legend (default is '')
- **fontsize** (*int, optional*) – Size in pixels of the font used for texts (default is 20)
- **width** (*float, optional*) – Width of the SVG drawing in vw units (default is 20.0)
- **height** (*float, optional*) – Height of the SVG drawing in vh units (default is 40.0)
- **bordercolor** (*str, optional*) – Color for lines and rects of the legend (default is 'black')
- **textcolor** (*str, optional*) – Color for texts of the legend (default is 'black')

- **dark** (*bool, optional*) – If True, the bordercolor and textcolor are set to white (default is False)

Return type
a string containing SVG text to display the graduated legend

Example

Creation of a SVG drawing to display a graduated legend. Input is prepared in the same way of the example provided for the `interMap.geojsonMap()` function:

```
import numpy as np
import pandas as pd
import plotly.express as px
from vois import svgMap, svgUtils

countries = svgMap.country_codes

# Generate random values and create a dictionary: key=countrycode, value=random in␣
↪[0.0,100.0]
d = dict(zip(countries, list(np.random.uniform(size=len(countries),low=0.0,high=100.
↪0))))

# Create a pandas dataframe from the dictionary
df = pd.DataFrame(d.items(), columns=['iso2code', 'value'])

svg = svgUtils.graduatedLegend(df, code_column='iso2code',
                               codes_selected=['IT', 'FR', 'CH'],
                               stroke_selected='red',
                               colorlist=px.colors.sequential.Viridis[::-1],
                               stdevnumber=2.0,
                               legendtitle='2020 Total energy consumption',
                               legendunits='KTOE per 100K inhabit.',
                               fontsize=18,
                               width=340, height=600)
display(HTML(svg))
```

`svgUtils.`**`svgLogo`**`()`

Creation of a simple SVG to display the logo of the European Commission

Return type
a string containing a SVG drawing that spans for 68 pixels in width and 44 pixels in height

Example

See example provided for the `svgUtils.svgTitle()` function

Note: This function is completely superseeded by the more complete examples of dashboard titles available using the `title.title` or `app.app`

`svgUtils.`**`svgTitle`**`(`*title='Dashboard title', subtitle1='Subtitle1', subtitle2='Subtitle2', xline=290*`)`

Creation of a simple title for a dashboard in SVG format

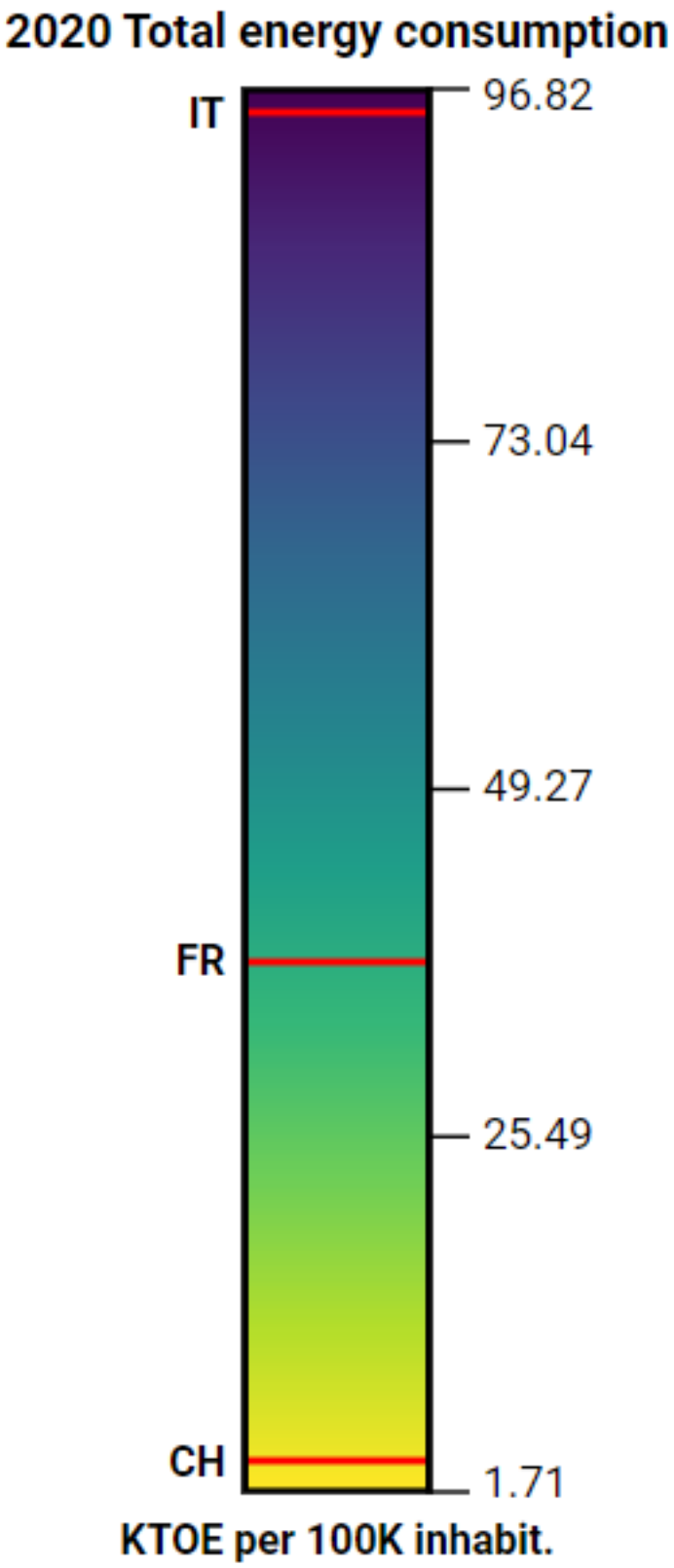


Fig. 3.28: Example of a graduatedLegend in SVG

Parameters

- **title** (*str, optional*) – Title of the dashboard (default is 'Dashboard title')
- **subtitle1** (*str, optional*) – First line of the subtitle (default is 'Subtitle1')
- **subtitle2** (*str, optional*) – Second line of the subtitle (default is 'Subtitle2')
- **xline** (*int, optional*) – X position of a vertical line to divide title from subtitles (default is 290)

Return type

a string containing a SVG drawing that spans 100% of the width and 46 pixels in height

Example

Example of a title and logo SVG:

```
from vois import svgUtils
from ipywidgets import HTML, widgets, Layout

outTitle = widgets.Output(layout=Layout(width='99%',                    height='64px
↪'))
outLogo  = widgets.Output(layout=Layout(width='1%', min_width='110px', height='82px
↪'))

outTitle.clear_output()
with outTitle:
    display(HTML(svgUtils.svgTitle()))

outLogo.clear_output()
with outLogo:
    display(HTML(svgUtils.svgLogo()))

display(widgets.HBox([outTitle,outLogo]))
```

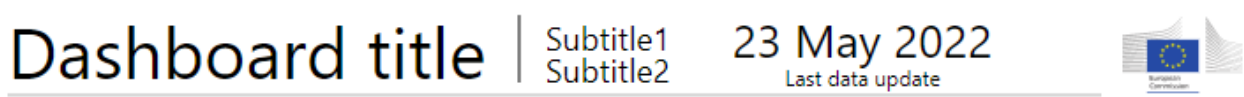


Fig. 3.29: Example of a simple svgTitle and svgLogo

Note: This function is completely superseeded by the more complete examples of dashboard titles available using the *title.title* or *app.app*

3.3.1.16 textpopup module

Map popup widget to display titles and texts in a geographic position on a ipyleaflet Map.

class textpopup.**textpopup**(*m*, *lat=0.0*, *lon=0.0*, *titles=[]*, *texts=[]*, *width=200*, *height=None*, *autoremovedelay=10.0*, *titlesbold=[]*, *titlefontsize=12*, *textsbold=[]*, *textfontsize=12*, *titlewidth=50*, *titlecolor='black'*, *textcolor='black'*, *lineheightfactor=1.1*)

Widget to vertically display a list of titles and texts strings as a popup on a ipyleaflet Map. Each couple of title and text occupies a row.

Parameters

- **m** (*instance of ipyleaflet.Map class*) – Map to which the popup has to be added
- **lat** (*float*) – Latitude where the popup has to be displayed (default is 0.0)
- **lon** (*float*) – Longitude where the popup has to be displayed (default is 0.0)
- **titles** (*list of strings*) – Strings to be displayed as title of each row (default is [])
- **texts** (*list of strings*) – Strings to be displayed as the content of each row (default is [])
- **width** (*int*) – Width of the popup in pixels (default is 200)
- **height** (*int*) – Height of the popup in pixel (default is None meaning that the height will be automatically calcolated)
- **autoremovedelay** (*float*) – Time in seconds for automatic remnoving of the popup (default is 0.0 which disables autoremove)
- **titlesbold** (*list of strings, optional*) – List of titles whose corresponding texts in the left column should be displayed using bold font (default is [])
- **titlefontsize** (*int, optional*) – Size in pixel of the font used for the titles (default is 12)
- **textsbold** (*list of strings, optional*) – List of titles whose corresponding texts in the right column should be displayed using bold font (default is [])
- **textfontsize** (*int, optional*) – Size in pixel of the font used for the texts (default is 12)
- **titlecolor** (*str, optional*) – Color to use for the titles (default is 'black')
- **textcolor** (*str, optional*) – Color to use for the texts (default is 'black')
- **lineheightfactor** (*float, optional*) – Factor to multiply to the font-size to calculate the height of each row (default is 1.5)

Example

Creation and display of a widget to display some textual information:

```
from ipywidgets import widgets, HTML, CallbackDispatcher
from ipyleaflet import Map
from IPython.display import display

from vois import textpopup
```

(continues on next page)

(continued from previous page)

```
m = Map(center=[43.66737, 12.5504], scroll_wheel_zoom=True, zoom=13)

t = None
def handle_interaction_popup(**kwargs):
    global t

    if kwargs.get('type') == 'click':
        lat = kwargs.get('coordinates')[0]
        lon = kwargs.get('coordinates')[1]

        textpopup.textpopup.removeAll(m)
        t = textpopup.textpopup(m, lat=lat, lon=lon, autoremovedelay=5.0,
                                width=340, height=None, titlewidth=70,
                                titles=['Pixel values', 'Class'],
                                texts=['(120,34,189)', 'Woodland and Shrubland␣
↪(incl. permanent crops)'],
                                titlesbold=[],
                                titlefontsize=11,
                                textsbold=['Pixel'],
                                textfontsize=11,
                                titlecolor='darkgreen',
                                textcolor='darkred')

m._interaction_callbacks = CallbackDispatcher()
m.on_interaction(handle_interaction_popup)

display(m)
```

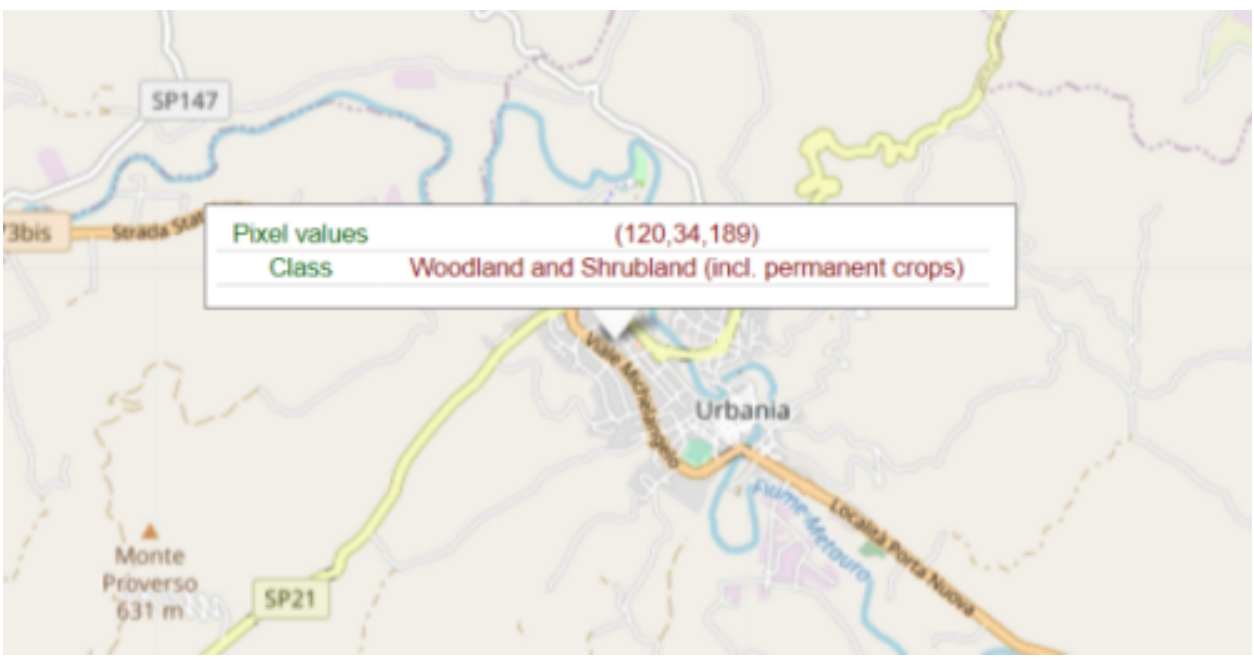


Fig. 3.30: Map popup widget for displaying textual information.

3.3.1.17 treemapPlotly module

Utility functions to prepare data for Plotly Treemap, Sunburst and Icicle plots.

`treemapPlotly.`**`createTreemapFromList`**(*nameslist=[]*, *rootName='Root'*, *separator='.'*, *valuefor={}*)

Preprocessing of a list of strings having a hierarchical structure (defined by a separator, e.g. '.'), in view of the display of a Plotly Treemap, Sunburst or Icicle chart. Each node of the tree will have a 'dimension' that influences the way to display it (the space occupied, or the size, etc.)

Parameters

- **nameslist** (*list of strings, optional*) – List of strings to preprocess. The hierarchical structure is defined by the separator character. Default is []
- **rootName** (*str, optional*) – Name to assign to the root node of the hierarchical structure (default is 'Root')
- **separator** (*str, optional*) – Separator character that defines the hierarchical structure (default is '.')
- **valuefor** (*dict, optional*) – Dictionary to assign a numerical value to each node of the tree (default is {})

Returns

- **a tuple of 3 elements** (*labels, parents, values*)
- *labels is a list of the names of the nodes*
- *parents is the list of the parents of each of the nodes*
- *values is the list of numerical values assigned to each of the node (and summed up in the parent-son relation)*
- *These 3 lists can be given as input to the plotly.graph_objects.Treemap/Sunburst/Icicle functions*

Example

Creation of a Treemap chart in Plotly using numerical data associated to JRC units and directorates:

```
import plotly.graph_objects as go
from vois import treemapPlotly

valuefor = {'JRC.A.1': 3.0, 'JRC.A.2': 5.0, 'JRC.B.1': 12.0, 'JRC.B.2': 7.0,
            'JRC.B.3': 3.0, 'JRC.C.1': 7.0, 'JRC.C.2': 2.0}

labels, parents, values = treemapPlotly.createTreemapFromList(['JRC.A','JRC.A.1',
                                                               'JRC.A.2','JRC.B.1',
                                                               'JRC.B.2','JRC.B.3',
                                                               'JRC.C.1','JRC.C.2',
↪],
                                                              rootName='JRC',
                                                              valuefor=valuefor)

fig = go.Figure()
fig.add_trace(go.Treemap(ids=labels, labels=labels, parents=parents, values=values,
                         branchvalues='total', maxdepth=3, root_color="lightgrey"))
```

(continues on next page)

(continued from previous page)

```
fig.update_layout(margin=dict(t=38, l=0, r=0, b=10), height=400,
                  hoverlabel=dict(bgcolor="#eee", align="left"),
                  title={'text': "JRC units", 'y':0.96, 'x':0.5,
                         'xanchor': 'center', 'yanchor': 'top'})

fig.show()
```

Fig. 3.31: Display of a Plotly Treemap extracted from a list of strings separated by '.'

3.3.1.18 urlOpen module

Utility functions to open a web page.

`urlOpen.`**`urlOpen`**(*url*, *output*, *target='_blank'*)

Opean a web page in another tab of the browser

Parameters

- **`url`** (*`str`*) – URL of the page to open
- **`output`** (*`instance of ipywidgets.Output() class`*) – Output widget where the javascript code to open the new page is executed
- **`target`** (*`str, optional`*) – Target of the open operation (default is '_blank' which means that the page will be opened in a new browser's tab)

Example

Open Google page in another tab of the browser:

```
from vois import urlOpen
from ipywidgets import widgets, Layout
from IPython.display import display

output = widgets.Output(layout=Layout(width='0px', height='0px'))
display(output)

urlOpen.urlOpen('https://www.google.com', output, target='_blank')
```

Note: If the dashboard is created using the *app.app* class, it is preferable to use the function *app.app.urlOpen()* that doesn't need the output parameter and uses the Output widget created inside the app instance itself (and invisible to the users)

3.3.1.19 urlUpdate module

Utility functions to update the URL of the page that launched the dashboard.

`urlUpdate.`**`urlUpdate`**(*url*, *output*)

Update the URL visualized in the top bar of the browser.

Parameters

url (*str*) – Partial url to add to the current browser's page key/values

Example

Add a key/value pair to the current browser URL:

```
from vois import urlUpdate
from ipywidgets import widgets, Layout
from IPython.display import display

output = widgets.Output(layout=Layout(width='0px', height='0px'))
display(output)

urlUpdate.urlUpdate('?test=3', output)
```

Note: If the dashboard is created using the *app.app* class, it is preferable to use the function *app.app.urlUpdate()* that doesn't need the output parameter and uses the Output widget created inside the app instance itself (and invisible to the users)

3.3.2 Vuetify modules

Vuetify modules of vois library contain classes that simplify the usage of ipyvuetify widgets inside a Jupyter notebook or a Voilà application.

3.3.2.1 app class

App class to easily define the structure of a typical Voilà dashboard.

class app.**app**(*title='Title of the dashboard', titlesvg='', titlesvgclass='pa-0 ma-0', titlecredits='', titlecredits2='', titlestyle='', titlespacestyle='width: 50px; min-width: 50px;', titleheight=70, totalheight=985, dark=False, backcolor='#efefef', titlewidth='600px', footercolor='lightgrey', backgroundimageurl=None, backgroundimageposition='center center', logourl='https://jeodpp.jrc.ec.europa.eu/services/shared/Notebooks/images/European_Commission.svg', logomaxwidth=80, logomaxheight=60, titletabs=['Tab 1', 'Tab 2', 'Tab 3'], titletabsdark=False, titletabsactive=0, titletabsstile='font-weight: bold;', titletabsactiveparameter=None, titletabscolor='#f8bd1a', footertext='2024 - Joint Research Centre', footerbuttons=['Home', 'About Us', 'Team', 'Services', 'Blog', 'Contact Us'], footerheight=68, footercopyright=True, footerdark=False, footercredits='', footercreditstooltip='', footercreditsurl='', sidepaneltitle='Settings', sidepanelwidth=400, sidepaneltext='', sidepanelcontent=[], sidepaneldark=False, sidepanelbackdark=False, minipanelicons=[], minipaneltooltips=[], minipanelopen=False, minipanellarge=True, minipanelbuttoncolor='black', minipaneliconscolor='black', onclicktab=None, onclickcredits=None, onclickcredits2=None, onclicklogo=None, onclickfooter=None, onclickminipanel=None, fullscreen=False*)

App class to easily define the basic structure of a typical Voilà dashboard.

An app instance displays a series of ipywidgets.Output() widgets to ease the usage of messages/dialog-boxes/etc, through these methods:

`snackbar()` `dialogMessage()` `dialogYesNo()` `dialogGeneric()` `dialogWaitOpen()` `dialogWaitClose()` `fab()` `downloadText()` `downloadBytes()` `urlOpen()` `urlParameter()` `urlParameter()`

The main content of the dashboard should be displayed in the **app.outcontent** Output widget that completely fills the empty space of the dashboard between the title bar and the footer bar.

Parameters

- **title** (*str, optional*) – Main title of the application to be displayed on the title bar (default is 'Title of the dashboard')
- **titlesvg** (*str, optional*) – SVG string to use as application title when the title string is empty (default is '')
- **titlesvgclass** (*str, optional*) – Class margins and padding to apply to the titlesvg drawing (default is 'pa-0 ma-0')
- **titlecredits** (*str, optional*) – Credits string to be displayed on the right side of the title bar (default is '')
- **titlecredits2** (*str, optional*) – Secondary credits string to be displayed on the right side of the title bar (default is '')
- **titlestyle** (*str, optional*) – CSS style to apply to the main title of the dashboard (default is '', an example could be: 'font-family: "Times New Roman", Times, serif; font-weight: bold; font-size: 22px;')

- **titlespacestyle** (*str, optional*) – CSS style to apply to the space at the left of the title (default is 'width: 50px; min-width: 50px;'). It can be useful to move the title more on the centre of the title bar, by providing a titlespacestyle like 'width: 200px;'
- **titleheight** (*int, optional*) – Height of the title bar in pixels (default is 70 pixels)
- **totalheight** (*int, optional*) – Total height in pixels of the page (default is 985 pixels which is coherent with a FullHD screen dimension)
- **dark** (*bool, optional*) – If True the title text color is settings.textcolor_dark, if False it is settings.textcolor_nodark (default is settings.dark_mode)
- **backcolor** (*str, optional*) – Background color of the title bar (default is settings.color_second)
- **titlewidth** (*str, optional*) – Width of the part of the title bar that contains the main title of the dashboard (default is '600px', other values could be, for instance: '50%')
- **footercolor** (*str, optional*) – Background color to use for the footer bar displayed in the bottom part of the screen (default is 'lightgrey')
- **backgroundimageurl** (*str, optional*) – URL of the optional image to display as background of the title bar (default is None)
- **backgroundimageposition** (*str, optional*) – Defines the way the backgroundimage in the title is cropped (default is 'center center'). See Vuetify.js v-img and CSS background-position documentation for help.
- **logourl** (*str, optional*) – URL of the image or SVG to use as a logo in the right end side of the title bar (default is European Commission logo)
- **logomaxwidth** (*int, optional*) – Maximum width in pixels of the logo image (default is 80 pixels)
- **logomaxheight** (*int, optional*) – Maximum height in pixels of the logo image (default is 60 pixels)
- **titletabs** (*list of strings, optional*) – List of tabs to be added to the title bar as the main visualization options of the dashboard (default is ['Tab 1', 'Tab 2', 'Tab 3'])
- **titletabsactive** (*int, optional*) – Index of the tab to activate at start (default is 0)
- **titletabsstile** (*str, optional*) – CSS style to apply to the tabs in the title bar (default is 'font-weight: bold;')
- **titletabsactiveparameter** (*str, optional*) – Name of the URL parameter to read for setting the activetab (default is None)
- **titletabscolor** (*str, optional*) – Color of the selected tab in the title tabs (default is settings.color_first)
- **titletabsdark** (*bool, optional*) – If True the text color of unselected tabs is settings.textcolor_dark, if False it is settings.textcolor_nodark (default is False)
- **footertext** (*str, optional*) – Text to display in the footer tab (default is '<current year> - Joint Research Centre')
- **footerbuttons** (*list of strings, optional*) – List of strings containing the caption of the buttons to display in the footer tab (default is ['Home', 'About Us', 'Team', 'Services', 'Blog', 'Contact Us'])
- **footerheight** (*int or float or str optional*) – Height of the footer bar. If an integer or a float is passed, the height is intended in pixels units, otherwise a string containing the units must be passed (example: '4vh'). Default is 68 for 68 pixels

- **footercopyright** (*bool, optional*) – If True adds the copyright symbol to the footer text (default is True)
- **footerdark** (*bool, optional*) – If True the footer text color is settings.textcolor_dark, if False it is settings.textcolor_nodark (default is settings.dark_mode)
- **footercredits** (*str, optional*) – Text for footer credits button (default is ‘’)
- **footercreditstooltip** (*str, optional*) – Tooltip for the footer credits button (default is ‘’)
- **footercreditsurl** (*str, optional*) – URL to open when the user clicks on the footer credits button (default is ‘’)
- **sidepaneltitle** (*str, optional*) – Title of the left side panel (default is ‘Settings’)
- **sidepanelwidth** (*int, optional*) – Width in pixels of the left side panel (default is 400). If the size is 0, the icon to open the sidepanel will not be added to the title bar
- **sidepaneltext** (*str, optional*) – Text to display in the left side panel (default is ‘’)
- **sidepanelcontent** (*list of ipywidgets, optional*) – List of ipywidgets object to display in the left side panel (default is [])
- **sidepaneldark** (*bool, optional*) – If True the title text color of the sidePanel is black, otehrwise is white (default is settings.dark_mode)
- **sidepanelbackdark** (*bool, optional*) – If True the background color of the sidePanel is black, otehrwise is white (default is settings.dark_mode)
- **minipanelicons** (*list of strings, optional*) – List of string containing the names of the icons to display in the minipanel that is displayed on the left side of the footer tab (default is [])
- **minipaneltooltips** (*list of strings, optional*) – List of tooltips to set for the icons of the minipanel (default is [])
- **minipanelopen** (*bool, optional*) – If True the minipanel is dopened on startup of the app (default is False)
- **minipanellarge** (*bool, optional*) – If True, the minipanel icons are displayed in the large version (default is True)
- **minipanelbuttoncolor** (*str, optional*) – Color of the ‘three vertical points icon’ that opens/closes the minipanel (default is the textcolor_notdark defined in the settings.py module)
- **minipaneliconscolor** (*str, optional*) – Color of the icons in the minipanel (default is the textcolor_notdark defined in the settings.py module)
- **onclicktab** (*function, optional*) – Python function to call when the user clicks on one of the tabs of the title bar. The function will receive a parameter of type string containing the text of the tab
- **onclickcredits** (*function, optional*) – Python function to call when the user clicks on the credits button on the title bar. The function will receive no parameters
- **onclickcredits2** (*function, optional*) – Python function to call when the user clicks on the secondary credits button on the title bar. The function will receive no parameters
- **onclicklogo** (*function, optional*) – Python function to call when the user clicks on the logo image on the title bar. The function will receive no parameters

- **onclickfooter** (*function, optional*) – Python function to call when the user clicks on one of the buttons of the footer bar. The function will receive a parameter of type string containing the text of the button
- **onclickminipanel** (*function, optional*) – Python function to call when the user clicks on one of the icons of the minipanel in the footer bar. The function will receive a parameter of type int containing the index of the icon
- **fullscreen** (*bool, optional*) – If True the app will be displayed in fullscreen mode (default is False). In fullscreen mode the app will occupy all the available space on the web page (the titlebar will be aligned on top, the footer bar will be aligned on the bottom of the page, and the outcontent will occupy all the intermediate space between the title bar and the footer bar, irrescpective of the value passed in the totalheight parameter), and the positioning of the elements will be fully responsive.

outcontent

This is the output widget where the content of the application can be displayed

To visually highlight the app.outcontent Output widget, this line of code can be executed:

```
g_app.outcontent.layout.border = '1px solid lightgrey'
```

After the execution of that line, the border of the g_app.outcontent will be visible. To reset the border to the empty line, this line of code can be executed:

```
g_app.outcontent.layout.border = ''
```

Type
ipywidgets.Output instance

Example

Creation of an app class to define the structure of a Voilà dashboard:

```
from vois.vuetify import app, settings
import ipyvuetify as v

# Change global settings
settings.dark_mode      = False
settings.color_second   = '#68aad2'
settings.color_first    = '#1c4056'
settings.button_rounded = False

# Click on a tab of the title
def on_click_tab(arg):
    g_app.snackbar(arg)

# Click on the credits text
def on_click_credits():
    g_app.snackbar('CREDITS')

# Click on the logo
def on_click_logo():
    g_app.snackbar('LOGO')
```

(continues on next page)

(continued from previous page)

```
# Click on the footer buttons
def on_click_footer(arg):
    g_app.snackbar(arg)

# Click on the footer minipanel
def on_click_minipanel(index):
    g_app.snackbar(str(index))

g_app = app.app(title='Energy consumption example dashboard',
                titlecredits='Created by Unit I.3',
                titlewidth='60%',
                footercolor='#1c4056',
                titletabs=['Chart', 'Table', 'Static Map', 'Dynamic Map'],
                titletabscolor='#60b3e8',
                dark=True,
                footerdark=True,
                footercredits='Data credits',
                footercreditstooltip='Eurostat - European Commission',
                footercreditsurl='https://ec.europa.eu/eurostat/data/database',
                backcolor='#1c4056',
                sidepaneltitle='Help',
                sidepaneltext='Lorem ipsum dolor sit amet, consectetur adipiscing␣
↪elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim␣
↪ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea␣
↪commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse␣
↪cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non␣
↪proident, sunt in culpa qui officia deserunt mollit anim id est laborum.',
                sidepanelcontent=[v.Icon(class_='pa-0 ma-0 ml-2', children=['mdi-
↪help'])],
                sidepaneldark=True,
                sidepanelbackdark=False,
                minipanelicons=['mdi-chart-bar', 'mdi-table-large',
                                'mdi-map-legend', 'mdi-file-chart-outline'],
                minipaneltooltips=['Download Chart', 'Download Table',
                                   'Download Map', 'Generate Report in Word'],
                minipanelbuttoncolor='white',
                onclickminipanel=on_click_minipanel,
                onclicktab=on_click_tab,
                onclickcredits=on_click_credits,
                onclicklogo=on_click_logo,
                onclickfooter=on_click_footer)

g_app.show()
```

contentAddPanel(*width*, *height*, *left*, *top*, *border='1px solid black'*, *backcolor='white'*, *name=''*, *number=None*, *title=''*, *titleround=False*, *titlewidth='8vw'*, *titleheight='2.4vh'*, *titlefontsize='1.8vh'*, *titlefontweight=500*, *icon=''*, *icontooltip=''*, *iconcolor='black'*, *icononclick=None*)

Adds a new panel to overlay on top of the outcontent output widget.

Parameters

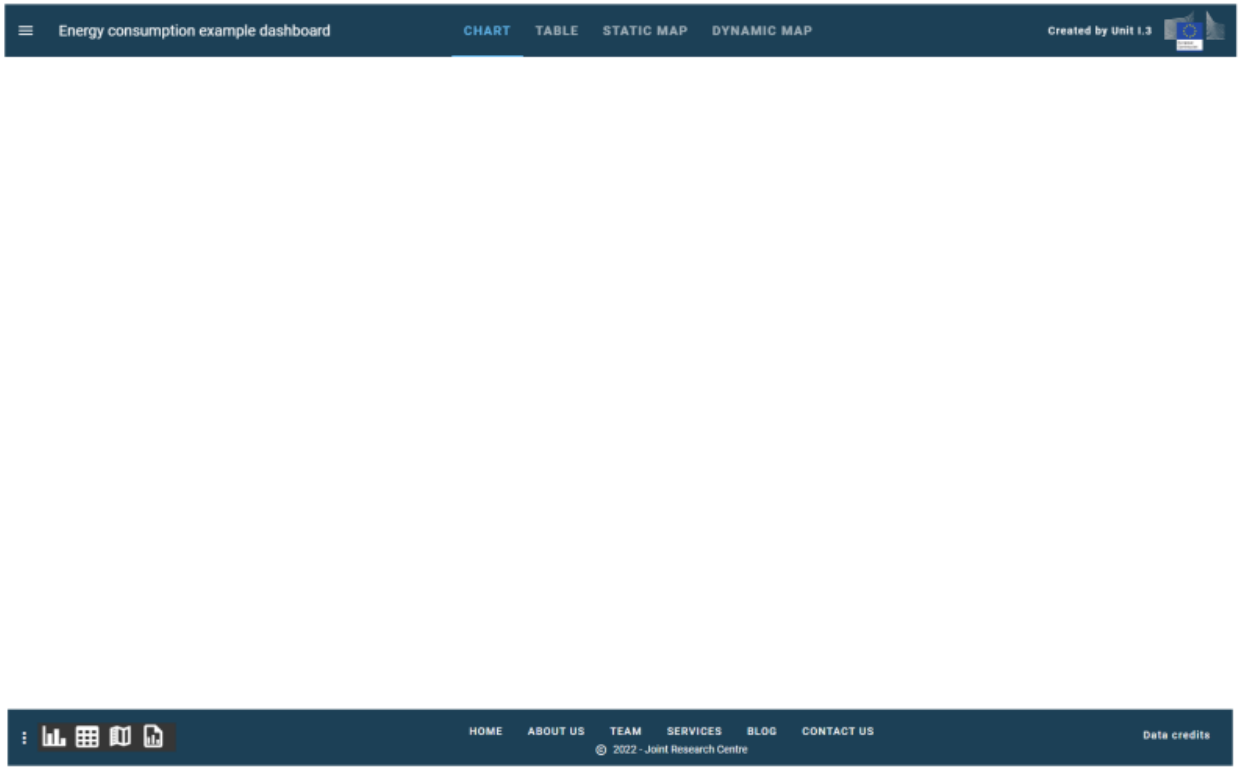


Fig. 3.32: App structure example

- **width** (*str*) – Width of the panel (can be in any CSS coordinates, examples: '300px', '40vw' or 'calc(100vw - 600px)')
- **height** (*str*) – Height of the panel (can be in any CSS coordinates, examples: '300px', '40vh' or 'calc(100vh - 400px)')
- **left** (*str*) – Position of the panel from the left border of the outcontent output widget (can be in any CSS coordinates, examples: '300px' or '10vw')
- **top** (*str*) – Position of the panel from the top border of the outcontent output widget (can be in any CSS coordinates, examples: '300px' or '10vh')
- **border** (*str, optional*) – Border of the panel (default is '1px solid black')
- **backcolor** (*str, optional*) – Background color of the panel (default is 'white'). Please be aware that colors different from 'white' work badly if used for panels that will contain ipywidgets or ipyvuetify widgets. Transparent colors (for example: #ffffff00) or semi-transparent colors (for example: #ffffff55) work well if the panel contains only one of the SVG charts of the vois library (svhHeatmap, svgBubblesChart, etc.)
- **name** (*str, optional*) – Name of the panel (default is '')
- **number** (*int, optional*) – Number to assign to the panel, to generate unique CSS class names (default is None which means that the number is automatically generated)
- **title** (*str, optional*) – Title to show in the top-left border of the panel (default is '')
- **titleround** (*bool, optional*) – If True the sides of the title are rounded (default is False)
- **titlewidth** (*str, optional*) – Width of the title area (default is '8vw')

- **titleheight** (*str, optional*) – Height of the title area (default is '2.4vh')
- **titlefontsize** (*str, optional*) – Size of the font to use to display the title (default is '1.8vh')
- **titlefontweight** (*int, optional*) – Weight of the font to use to display the title (default is 500)
- **icon** (*str, optional*) – Name of an icon to display in the title bar of the panel (default is None)
- **icontooltip** (*str, optional*) – Tooltip to show when hover on the icon (default is '')
- **iconcolor** (*str, optional*) – Color of the icon (default is 'black')
- **iononclick** (*function, optional*) – Python function to call when the user clicks on the icon. The function will receive as parameter the name of the panel (default is None)

contentBackground(*imageUrl=None*)

Sets the background image for the outcontent output widget

Parameters

imageUrl (*str, optional*) – URL of the optional image to display as background of the outcontent output widget (default is None)

contentResetPanels()

Resets the list of panels to display on top of the outcontent output widget

contentSetPanel(*name, width, height, left, top, border='1px solid black', backcolor='white', titleround=False, titlewidth='8vw', titleheight='2.4vh', titlefontsize='1.8vh', titlefontweight=500, icon='', icontooltip='', iconcolor='black', iononclick=None*)

Change position, sizing and colors of a panel given its name.

Parameters

- **name** (*str*) – Name of the panel to modify
- **width** (*str*) – Width of the panel (can be in any CSS coordinates, examples: '300px', '40vw' or 'calc(100vw - 600px)')
- **height** (*str*) – Height of the panel (can be in any CSS coordinates, examples: '300px', '40vh' or 'calc(100vh - 400px)')
- **left** (*str*) – Position of the panel from the left border of the outcontent output widget (can be in any CSS coordinates, examples: '300px' or '10vw')
- **top** (*str*) – Position of the panel from the top border of the outcontent output widget (can be in any CSS coordinates, examples: '300px' or '10vh')
- **border** (*str, optional*) – Border of the panel (default is '1px solid black')
- **backcolor** (*str, optional*) – Background color of the panel (default is 'white'). Please be aware that colors different from 'white' work badly if used for panels that will contain ipywidgets or ipyvuetify widgets. Transparent colors (for example: #ffffff00) or semi-transparent colors (for example: #ffffff55) work well if the panel contains only one of the SVG charts of the vois library (svhHeatmap, svgBubblesChart, etc.)
- **titleround** (*bool, optional*) – If True the sides of the title are rounded (default is False)
- **titlewidth** (*str, optional*) – Width of the title area (default is '8vw')
- **titleheight** (*str, optional*) – Height of the title area (default is '2.4vh')

- **titlefontsize** (*str, optional*) – Size of the font to use to display the title (default is '1.8vh')
- **titlefontweight** (*int, optional*) – Weight of the font to use to display the title (default is 500)
- **icon** (*str, optional*) – Name of an icon to display in the title bar of the panel (default is None)
- **icontooltip** (*str, optional*) – Tooltip to show when hover on the icon (default is '')
- **iconcolor** (*str, optional*) – Color of the icon (default is 'black')
- **iononclick** (*function, optional*) – Python function to call when the user clicks on the icon. The function will receive as parameter the name of the panel (default is None)

dialogGeneric(**args*, ***kwargs*)

Display a dialogGeneric. See *dialogGeneric()* for the list of parameters.

dialogMessage(**args*, ***kwargs*)

Display a dialogMessage. See *dialogMessage()* for the list of parameters.

dialogWaitClose()

Close the dialogWait. See *close()*.

dialogWaitOpen(**args*, ***kwargs*)

Open a dialogWait. See *dialogWait()* for the list of parameters.

dialogYesNo(**args*, ***kwargs*)

Display a dialogYesNo. See *dialogYesNo()* for the list of parameters.

display(*arg*)

Display something in the service Output of this application

downloadBytes(*bytesobj*, *fileName='download.bin'*)

Direct download of an array of bytes.

Parameters

- **bytesobj** (*bytes-like object*) – Bytes to write in the file to be downloaded
- **fileName** (*str, optional*) – Name of the file that is to be downloaded (default is 'download.bin')

Example

If g_app is an instance of the app class, this code will download a binary file containing the passed bytes:

```
g_app.downloadBytes(b'ajgh lkjhl ')
```

downloadText(*textobj*, *fileName='download.txt'*)

Direct download of a .txt file containing a string.

Parameters

- **textobj** (*str*) – Text to write in the file to be downloaded
- **fileName** (*str, optional*) – Name of the file that is to be downloaded (default is 'download.txt')

Example

If g_app is an instance of the app class, this code will download a text file containing the passed string:

```
g_app.downloadText('aaa bbb ccc')
```

fab(**args*, ***kwargs*)

Open a fab button. See *fab()* for the list of parameters.

setActiveTab(*index*)

Set the active tab of the title bar of the app

show()

Display the app

snackbar(**args*, ***kwargs*)

Display a message in a snackbar. See *snackbar()* for the list of parameters.

urlOpen(*url*, *target='_blank'*)

Open a web page in another tab.

Parameters

- **url** (*str*) – URL of the page to be opened
- **target** (*str, optional*) – Target of the open operation (default is '_blank' which means that the page will be opened in a new browser's tab)

Example

If g_app is an instance of the app class, this code will open a new tab in the browser:

```
g_app.urlOpen('https://www.google.com')
```

urlParameter(*parameterName*, *parameterDefaultValue=''*)

Read the parameters passed on the URL.

Parameters

- **parameterName** (*str*) – name of the parameter to read from the URL that launched the Voilà dashboard
- **parameterDefaultValue** (*str, optional*) – Default value of the parameter, in case it is not present in the URL that launched the Voilà dashboard (default is '')

Example

If g_app is an instance of the app class, this code print the value of an URL parameter:

```
print(g_app.urlParameter('activetab'))
```

urlUpdate(*url*)

Update the URL visualized in the top bar of the browser.

Parameters

url (*str*) – Partial url to add to the current browser's page key/values

Example

If g_app is an instance of the app class, this code will add a key=value to the URL of the application:

```
g_app.urlUpdate('?test=3')
```

3.3.2.2 basemaps widget

Widget to select the basemap to visualise on a ipyleaflet Map

class basemaps.**basemaps**(*m*, *color='#f8bd1a'*, *dark=False*, *width=320*, *height=650*, *addBDAPbasemaps=True*, *removeBasemaps=[]*, *rootName='Basemaps'*, *onchange=None*)

Treeview widget to select a basemap for an ipyleaflet map.

Parameters

- **m** (*ipyleaflet.Map instance*) – Map instance on which the selected basemap has to be set as backdrop layer
- **color** (*str, optional*) – Color to use for the widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the widget will have a dark background (default is settings.dark_mode)
- **width** (*int, optional*) – Width of the widget in pixels (default is 320)
- **height** (*int, optional*) – Height of the widget in pixels (default is 650)
- **addBDAPbasemaps** (*bool, optional*) – If True the treeview will contain also some BDAP layers selectable as basemaps (default is True)
- **removeBasemaps** (*list of str, optional*) – List of basemaps names to be removed from the widget (default is [])
- **rootName** (*str, optional*) – Name to use as the root node of the basemaps treeview (default is 'Basemaps')
- **onchange** (*function, optional*) – Python function to call when the user selects a different basemap. The function will receive no parameters. (default is None)

Example

Creation of a basemap selection widget:

```
from jeodpp import inter, imap
from ipywidgets import widgets, Layout

from vois.vuetify import basemaps

height = 650

m = imap.Map(layout=Layout(height='%dpx'%height))

b = basemaps.basemaps(m, height=height, dark=False)

display(widgets.HBox([b.draw(),m]))
```

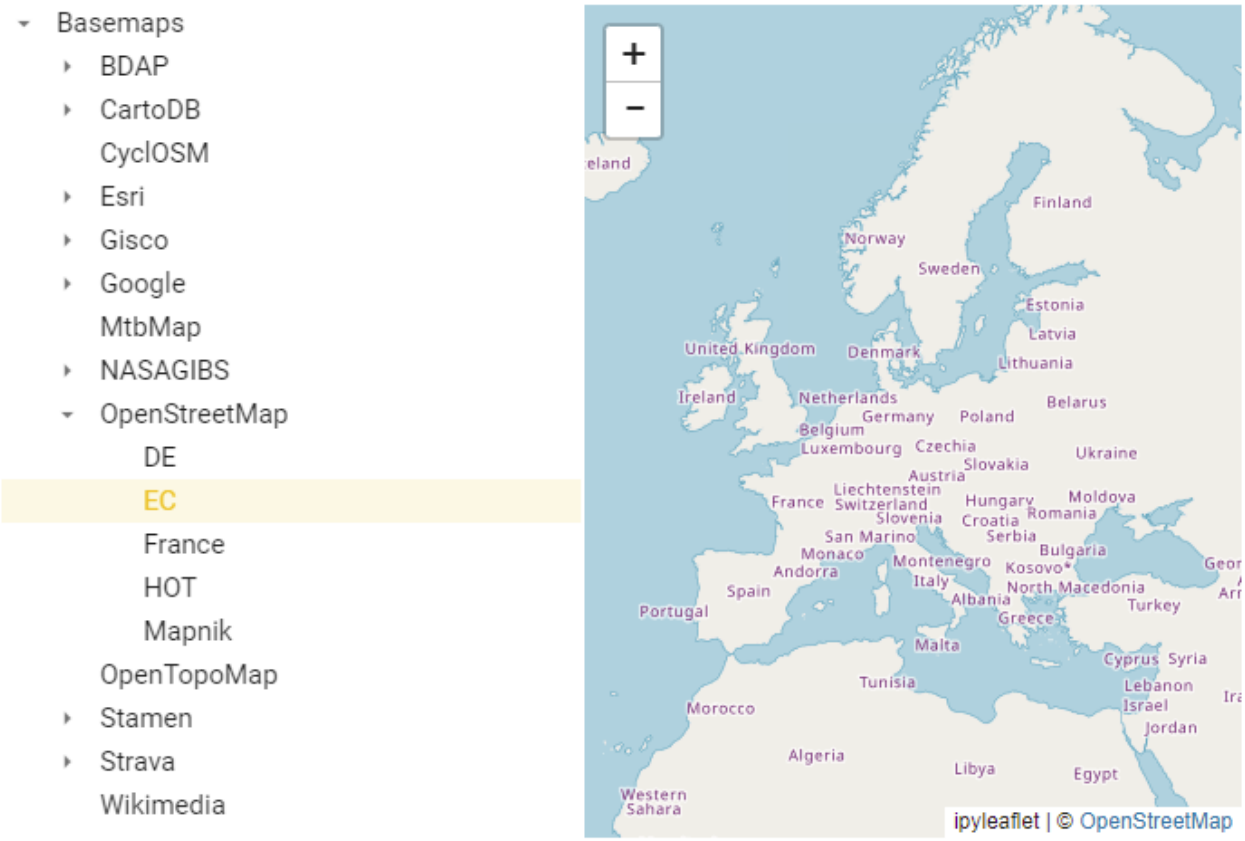


Fig. 3.33: Example of a basemaps selection widget

property current_layer

Get the currently select layer (instance of ipyleaflet.leaflet.TileLayer class or ipyleaflet.leaflet.LayerGroup class)

draw()

Returns the ipyvuetify object to display (the internal v.Card containing the treeview)

reset()

Set the default basemap (OpenStreetMap.EC)

property value

Get/Set the active basemap name.

Returns
name – Name of the basemap

Return type
str

Example

Programmatically select one of the basemaps and print the value selected:

```
b.value = 'Esri.WorldImagery'
print(b.value)
```

3.3.2.3 button widget

Button widget to call a python function when clicked.

class button.**button**(*text*, *onclick=None*, *argument=None*, *width=100*, *height=36*, *selected=False*, *disabled=False*, *tooltip=''*, *large=False*, *xlarge=False*, *small=False*, *xsmall=False*, *outlined=False*, *textweight=500*, *href=None*, *target=None*, *onlytext=False*, *textcolor=None*, *class_='pa-0 ma-0'*, *icon=None*, *iconlarge=False*, *iconsmall=False*, *iconleft=False*, *iconcolor='black'*, *autoselect=False*, *dark=False*, *rounded=True*, *tile=False*, *colorselected='#f8bd1a'*, *colorunselected='#efefef'*, *ondblclick=None*)

Button widget to call a python function when clicked.

Parameters

- **text** (*str*) – Test string to be displayed on the button widget
- **onclick** (*function, optional*) – Python function to call when the user clicks on the button. The function will receive as parameter the value of the argument (default is None)
- **ondblclick** (*function, optional*) – Python function to call when the user double-clicks on the button. The function will receive as parameter the value of the argument (default is None)
- **argument** (*any, optional*) – Argument to be passed to the onclick function when user click on the label (default is None)
- **width** (*int, optional*) – Width of the button widget in pixels (default is 100)
- **height** (*int, optional*) – Height of the button widget in pixels (default is 36)
- **selected** (*bool, optional*) – Flag to show the button as selected (default is False)

- **disabled** (*bool, optional*) – Flag to show the button as disabled (default is False)
- **tooltip** (*str, optional*) – Tooltip text to show when the user hovers on the button (default is '')
- **large** (*bool, optional*) – Flag that sets the large version of the button (default is False)
- **xlarge** (*bool, optional*) – Flag that sets the xlarge version of the button (default is False)
- **small** (*bool, optional*) – Flag that sets the small version of the button (default is False)
- **xsmall** (*bool, optional*) – Flag that sets the xsmall version of the button (default is False)
- **outlined** (*bool, optional*) – Flag to show the button as outlined (default is False)
- **textweight** (*int, optional*) – Weight of the text to be shown in the label (default is 500, Bold is any value greater or equal to 500)
- **href** (*str, optional*) – URL to open when the button is clicked (default is None)
- **target** (*str, optional*) – Designates the target attribute (where the URL page is opened, for instance: '_blank' to open it in a new browser tab). This should only be applied when using the href parameter (default is None)
- **onlytext** (*bool, optional*) – If True, the button will contain only the text (default is False)
- **textcolor** (*str, optional*) – Color used for the button text (default is None)
- **icon** (*str, optional*) – Name of the icon to display aside the text of the label (default is None)
- **iconlarge** (*bool, optional*) – Flag that sets the large version of the icon (default is False)
- **iconsmall** (*bool, optional*) – Flag that sets the small version of the icon (default is False)
- **iconleft** (*bool, optional*) – Flag that sets the position of the icon to the left of the text of the label (default is False)
- **iconcolor** (*str, optional*) – Color of the icon (default is 'black')
- **autoselect** (*bool, optional*) – If True, the button becomes selected when clicked (default is False)
- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)
- **rounded** (*bool, optional*) – Flag to display the button with rounded corners (default is the value of settings.button_rounded)
- **tile** (*bool, optional*) – Flag to remove the button small border (default is False)
- **colorselected** (*str, optional*) – Color used for the button when it is selected (default is settings.color_first)
- **colorunselected** (*str, optional*) – Color used for the button when it is not selected (default is settings.color_second)

Note: All the icons from https://materialdesignicons.com/ site can be used, just by prepending 'mdi-' to their name.

All the free icons from https://fontawesome.com/ site can be used, just by prepending 'fa-' to their name.

Example

Creation and display of a some button widgets playing with the parameters:

```
from vois.vuetify import settings, button

def onclick(arg=None):
    if arg==1: b1.selected = not b1.selected
    if arg==2: b2.selected = not b2.selected
    else:      b3.selected = not b3.selected

b1 = button.button('Test button 1', textweight=300, onclick=onclick, argument=1,
                   width=150, height=36,
                   tooltip='Tooltip for button 1', selected=False, rounded=True,
                   icon='mdi-car-light-high', iconcolor='black')

b2 = button.button('Test button 2', textweight=450, onclick=onclick, argument=2,
                   width=150, height=48,
                   tooltip='Tooltip for button 2', selected=True, rounded=False)

b3 = button.button('Test button 3', textweight=450, onclick=onclick, argument=3,
                   width=150, height=38,
                   textcolor=settings.color_first,
                   tooltip='Tooltip for button 3', outlined=True, rounded=True)

b4 = button.button('Contacts', onlytext=True, textcolor=settings.color_first,
                   width=150, height=28,
                   href='https://ec.europa.eu/info/contact_en', target="_blank",
                   tooltip='Open a URL')

display(b1.draw())
display(b2.draw())
display(b3.draw())
display(b4.draw())
```

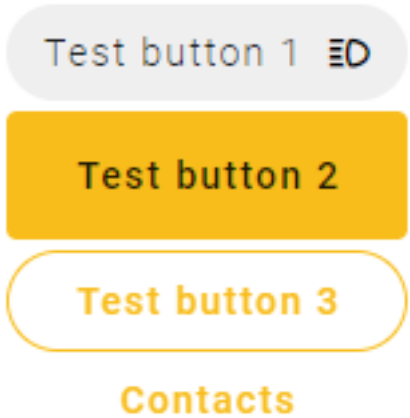


Fig. 3.34: Example of a 4 button widgets with different display modes.

property disabled

Get/Set the disabled state of the button widget.

Returns
disabled status – True if the button is disabled, False otherwise

Return type
bool

draw()

Returns the ipyvuetify object to display (the internal v.Html containing a v.Btn widget as its only child)

property selected

Get/Set the selected state of the button widget.

Returns
selected status – True if the button is selected, False otherwise

Return type
bool

Example

Programmatically select a button:

```
b.selected = True
print(b.selected)
```

setIcon(*iconname*)

Change the icon for the button

Example

Creation of a button and programmatically change of its icon:

```
from vois.vuetify import settings, button

b = button.button('Test button', textweight=450, width=150, height=46,
                  selected=True, rounded=True,
                  icon='mdi-menu-open', iconcolor='black', iconlarge=True)
display(b.draw())
b.setIcon('mdi-menu')
```

setText(*newtext*)

Change the text for the button

Example

Creation of a button and programmatically change of its icon:

```
from vois.vuetify import settings, button

b = button.button('Test button', textweight=450, width=250, height=46,
                  selected=True, rounded=True)
display(b.draw())
b.setText('New button text')
```

3.3.2.4 card widget

Simple card with title, subtitle and image.

class card.**card**(***kwargs: Any*)

Simple card displaying title, subtitle and an image.

Parameters

- **width** (*str, optional*) – Width of the card (default is '400px')
- **color** (*str, optional*) – Background color of the card (default is 'white')
- **dark** (*bool, optional*) – If True the title and subtitle texts are displayed in white color, if False they are displayed in black (default is False)
- **ripple** (*bool, optional*) – If True the click on the card is highlighted (default is False)
- **elevation** (*int, optional*) – Elevation of the card over the background of the page (default is 3)
- **title** (*str, optional*) – Title of the card (default is 'Title')
- **subtitle** (*str, optional*) – Subtitle of the card (default is 'Subtitle')
- **icon** (*str, optional*) – URL of the image to display as an icon before the card title (default is '')
- **iconsize** (*str, optional*) – Size of the area where the icon is displayed (default is '32px')
- **image** (*str, optional*) – URL of the image to display in the right side of the card (default is '')
- **imagesize** (*str, optional*) – Size of the area where the image is displayed (default is '190px')
- **on_click** (*function, optional*) – Python function to call when the user clicks on the card. The function will receive no parameters. (default is None)
- **argument** (*any, optional*) – Argument to pass to the on_click python function (default is None)
- **responsive** (*bool, optional*) – If True, the font size is automatically changed according to the page size (default is False)
- **fontsizemultiplier** (*float, optional*) – Multiply factor for changing the standard size of the font used for title and subtitle (default is 1.0)

- **backgroundimageurl** (*str, optional*) – URL of the optional image to display as background of the card (default is '')
- **tooltip** (*str, optional*) – Text to display as tooltip of the whole card (default is '')
- **titletooltip** (*str, optional*) – Text to display as tooltip of the card title (default is '')
- **focusedopacity** (*float, optional*) – Opacity of the card background when the card is clicked (has focus). Default is 0.1
- **textcolor** (*str, optional*) – Color of the text (default is 'black')
- **titleweight** (*int, optional*) – Font weight for the title (default is 700)
- **subtitleweight** (*int, optional*) – Font weight for the subtitle (default is 400)

Example

Creation of a card to display text and an image:

```
from vois.vuetify import card
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def on_click():
    with output:
        print('clicked!')

subtitle = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod␣
↪tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis␣
↪nostrud exercitation ullamco laboris nisi...'
width = 600

c = card.card(elevation=5, width='600px', title='Sample dataset', subtitle=subtitle,
              image='https://cdn.vuetifyjs.com/images/cards/halcyon.png', on_
↪click=on_click)

display(c)
display(output)
```

3.3.2.5 cardsGrid widget

Cards with title, subtitle and image displayed in rows and columns

class cardsGrid.**cardsGrid**(***kwargs: Any*)

Cards with title, subtitle and image displayed in a grid of rows and columns.

Parameters

- **cards** (*list of json element, one for each card to display, optional*) – Each of the json elements must have this structure: { "title": "", "subtitle": "", "image": ""}, optional tags are "color", "imagesize", "icon" and "iconsize", "titletooltip"
- **width** (*str, optional*) – Width of the cards (default is '400px')

Fig. 3.35: Example of a card with text and an image

- **height** (*str, optional*) – Heigth of the cards (default is '' which means that each card has its own height defined by its content)
- **cols** (*int, optional*) – Horizontal column span [1,12] for each of the card (default is 6)
- **color** (*str, optional*) – Background color of the cards (default is 'white')
- **dark** (*bool, optional*) – If True the title and subtitle texts are displayed in white color, if False they are displayed in black (default is False)
- **ripple** (*bool, optional*) – If True the click on the card is highlighted (default is False)
- **iconsize** (*str, optional*) – Size of the area where the icon is displayed (default is '32px')
- **imagesize** (*str, optional*) – Size of the area where the image is displayed (default is '190px')
- **on_click** (*function, optional*) – Python function to call when the user clicks on the card. The function will receive as parameter the index of the clicked card. (default is None)
- **responsive** (*bool, optional*) – If True, the font size is automatically changed according to the page size (default is False)
- **fontsizemultiplier** (*float, optional*) – Multiply factor for changing the standard size of the font used for title and subtitle (default is 1.0)
- **tooltipwidth** (*str, optional*) – Max width of the tooltip window to open on hover on the cards title (default is '600px')

Example

Creation of a cards grid to display text and an image:

```
from vois.vuetify import cardsGrid
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def on_click(index):
```

(continues on next page)

(continued from previous page)

```
    with output:
        print(index)

cards = [
   { "title": "title0", "subtitle": "subtitle0", "image": "https://cdn.vuetifyjs.
↪com/images/cards/sunshine.jpg", "titletooltip": "Example of card title tooltip" },
   { "title": "title1", "subtitle": "subtitle1", "image": "https://cdn.vuetifyjs.
↪com/images/cards/road.jpg" },
   { "title": "title2", "subtitle": "subtitle2", "image": "https://cdn.vuetifyjs.
↪com/images/cards/plane.jpg" },
   { "title": "title3", "subtitle": "subtitle3", "image": "https://cdn.vuetifyjs.
↪com/images/cards/house.jpg" }
]

g = cardsGrid.cardsGrid(cards=cards, cols=6, width='350px', imagesize='200px', on_
↪click=on_click)

display(g)
display(output)
```

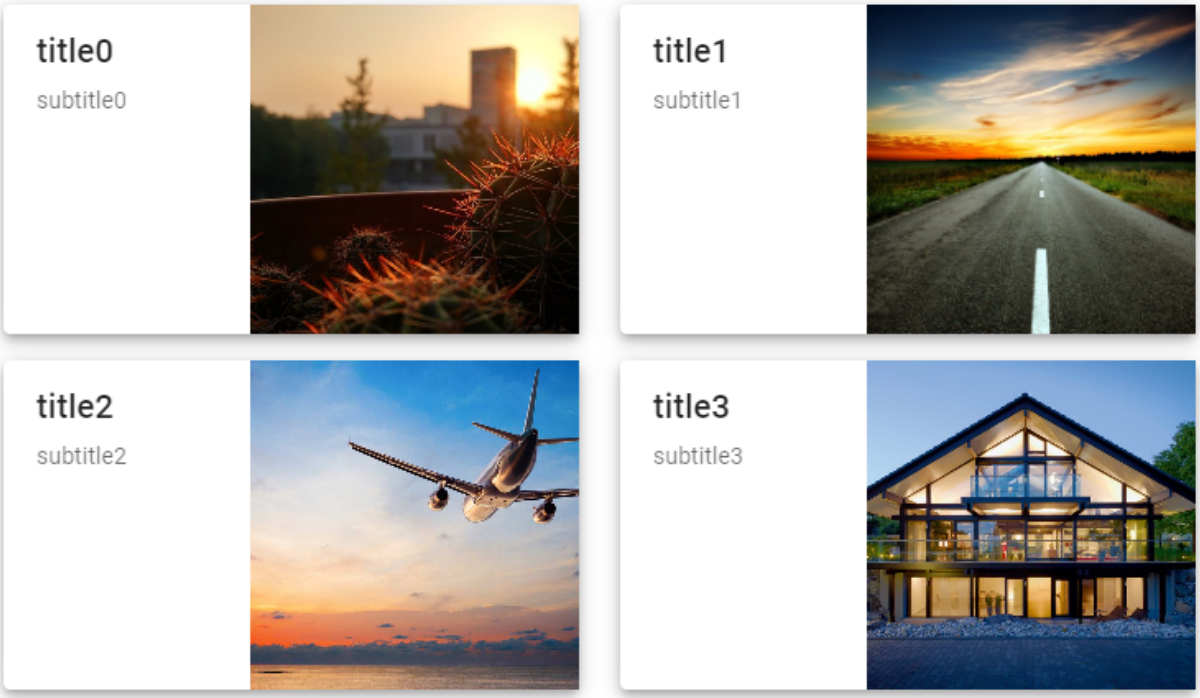


Fig. 3.36: Example of a cardsGrid to display multiple cards containing texts and an images

3.3.2.6 colorPicker widget

Input widget to select a color

class colorPicker.**colorPicker**(*color='#FF0000', dark=False, width=40, height=30, rounded=False, canvas_height=100, show_canvas=True, show_mode_switch=True, show_inputs=True, show_swatches=True, swatches_max_height=164, text='', textweight=400, onchange=None, argument=None, offset_x=False, offset_y=True, disabled=False*)

Input widget to select a color.

Parameters

- **color** (*str, optional*) – Initial color selected on the widget expressed in hexadecimal format '#RRGGBB' (default is '#FF0000')
- **dark** (*bool, optional*) – If True, the popup color selection will have a dark background (default is settings.dark_mode)
- **width** (*int, optional*) – Width of the widget in pixels (default is 40)
- **height** (*int, optional*) – Height of the widget in pixels (default is 30)
- **rounded** (*bool, optional*) – If True the color widget is displayed as a round button (default is False)
- **canvas_height** (*int, optional*) – Height of the canvas displayed on top of the popup window to select the colors (default is True)
- **show_canvas** (*bool, optional*) – If True the popup window will show the color canvas (default is True)
- **show_mode_switch** (*bool, optional*) – If True the popup window will show mode switch control among RGB, HSL and HAX (default is True)
- **show_inputs** (*bool, optional*) – If True the popup window will show the input field for the color components (default is True)
- **show_swatches** (*bool, optional*) – If True the popup window will show the color swatches (default is True)
- **swatches_max_height** (*int, optional*) – Height in pixels of the swatches area in the popup window (default is 164)
- **text** (*str, optional*) – Text to display in the color button (default is '')
- **textweight** (*int, optional*) – Weight of the text to be shown in the button (default is 400, Bold is any value greater or equal to 500)
- **onchange** (*function, optional*) – Python function to call when the user selects a different color. The function will receive the argument parameter. If the argument is None, the function will receive no parameters. (default is None)
- **argument** (*any, optional*) – Argument to be passed to the onchange function (default is None)
- **offset_x** (*bool, optional*) – If True the popup window will be opened on the right of the color button (default is False)
- **offset_y** (*bool, optional*) – If True the popup window will be opened on the bottom of the color button (default is True)
- **disabled** (*bool, optional*) – True if the selection of the color is disabled, False otherwise (default is False)

Example

Creation of a color picker widget to select a color:

```
from vois.vuetify import colorPicker
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange():
    with output:
        print('Changed to', c.color)

c = colorPicker.colorPicker(color='#00AAFF',
                            width=30, height=30,
                            rounded=False,
                            onchange=onchange,
                            offset_x=True,
                            offset_y=False)

display(c.draw())
display(output)
```

property color

Get/Set the selected color.

Returns
c – color currently selected

Return type
str

Example

Programmatically change the color:

```
picker.color = '#00FF00'
print(picker.color)
```

property disabled

Get/Set the disabled state of the widget.

Returns
flag – True if the widget is disabled, False otherwise

Return type
bool

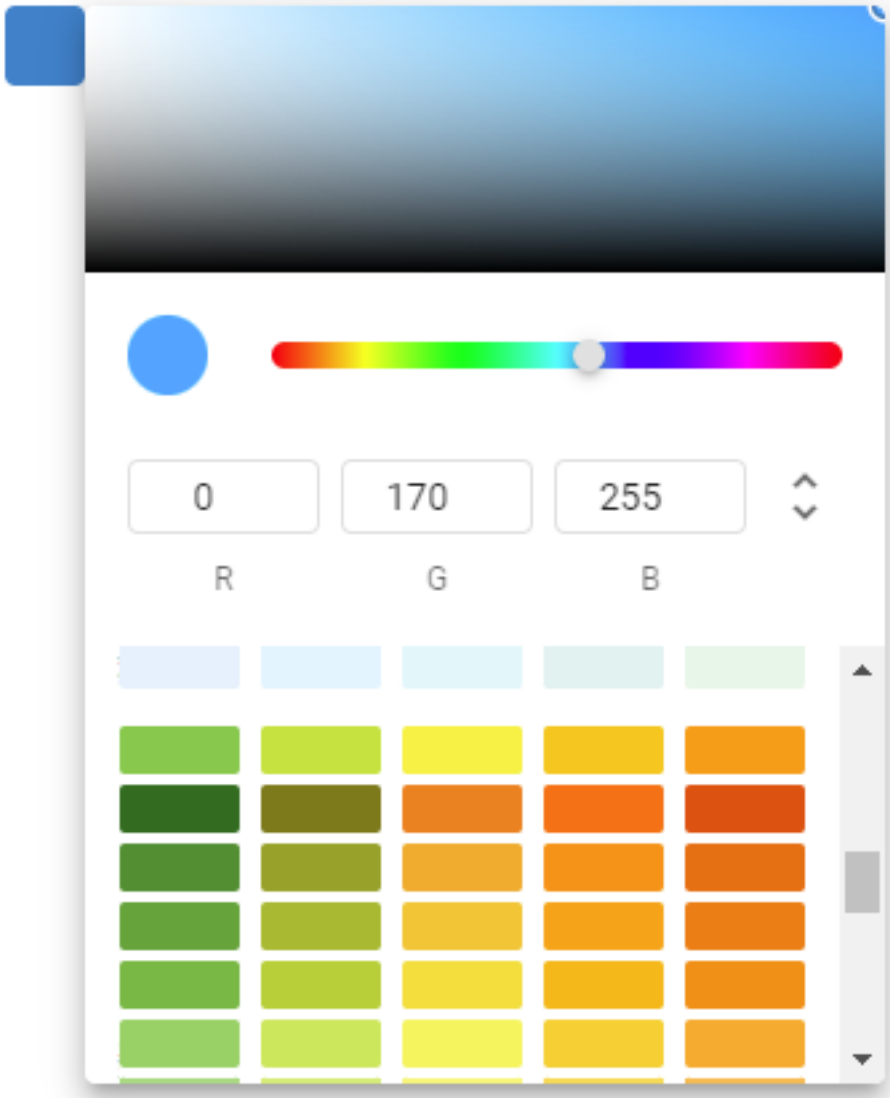


Fig. 3.37: Example of a colorPicker to select a color

Example

Programmatically change the date:

```
picker.disabled = True
print(picker.disabled)
```

draw()

Returns the ipyvuetify object to display (the internal v.Menu)

3.3.2.7 datatable widget

Display of a Pandas DataFrame in a data-table widget.

class datatable.**datatable**(***kwargs: Any*)

Display of a Pandas DataFrame in a data-table widget.

Parameters

- **data** (*Pandas DataFrame, optional*) – Pandas DataFrame to be displayed (default is pd.DataFrame() empty DataFrame)
- **height** (*str, optional*) – Height of the data-table widget (default is '400px')
- **on_click** (*function, optional*) – Python function to call when the user clicks on one of the rows of the data-table. The function will receive a parameter of type dict containing all the column names as keys and the clicked row data as values
- **color** (*str, optional*) – Color to use for the display of alert and warning messages (for instance if no records are present in input DataFrame) (default is 'error')
- **dark** (*bool, optional*) – If True, the error and warning messages are displayed in white color, if False they are displayed in black (default is False)
- **searchshow** (*bool, optional*) – If True, on top of the table a search field will be displayed, allowing for search (default is False)
- **search** (*str, optional*) – Initial search string to be displayed in the search field (default is ''). If searchshow is False, this argument will not be used
- **title** (*str, optional*) – In case searchshow is True, a Title can be shown on top of the datatable (default is '')
- **icon** (*str, optional*) – In case searchshow is True, an icon can be shown on the right of the datatable title (default is '')
- **icon_tooltip** (*str, optional*) – Tooltip string to be used for the icon (default is '')
- **icon_color** (*str, optional*) – Color of the icon (default is 'grey')
- **icon_disabled** (*bool, optional*) – If True, the icon will be disabled (default is False)
- **font_size** (*str, optional*) – Font size to use for the rows and headers of the datatable (default is '14px')
- **font_size_title** (*str, optional*) – Font size to use for the title of the datatable (default is '16px')
- **on_icon** (*function, optional*) – Python function to call when the user clicks on the icon on the top of the data-table. The function will receive no parameters

- **unsortable_columns** (*list of str, optional*) – List of names of columns that must not be sortable in the datatable (please note that the DataFrame column names will be displayed in capital letters in the datatable: use in this list the original column names of the DataFrame and not the capitalized names)

Example

Creation of a Pandas DataFrame from the 'Our World In Data' dataset on Covid-19 daily data and display of last 100 days for Italy:

```
from vois.vuetify import datatable
import pandas as pd
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

df = pd.read_csv('https://raw.githubusercontent.com/owid/covid-19-data/master/
↪public/data/owid-covid-data.csv')

def on_click(data):
    output.clear_output()
    with output:
        print(data)

df = df[df['location']=='Italy']
d = datatable.datatable(data=df.tail(100), height='500px', on_click=on_click)

display(d)
display(output)
```

3.3.2.8 datePicker widget

Input widget to select a date

class datePicker.**datePicker**(*date=None, label='', dark=False, width=88, color='#f8bd1a', show_week=False, onchange=None, offset_x=False, offset_y=True, disabled=False, mindate=None, maxdate=None*)

Input widget to select a date.

Parameters

- **date** (*str, optional*) – Initial date selected on the input widget expressed in format 'YYYY-MM-DD' (default is None, corresponding to the today date)
- **label** (*str, optional*) – Label to be displayed inside the widget (default is '')
- **dark** (*bool, optional*) – If True, the popup date selection will have a dark background (default is settings.dark_mode)
- **width** (*int, optional*) – Width of the widget in pixels (default is 88)
- **color** (*str, optional*) – Color to use for the widget and the header of the popup window (default is settings.color_first)

index	iso_code	continent	location	date	total_cases	new_cases	new_cases_sm
84749	ITA	Europe	Italy	2022-02-14	12134451	28776	67301.857
84750	ITA	Europe	Italy	2022-02-15	12205474	71023	62815.286
84751	ITA	Europe	Italy	2022-02-16	12265343	59869	59701
84752	ITA	Europe	Italy	2022-02-17	12323398	58055	57109.571
84753	ITA	Europe	Italy	2022-02-18	12377098	53700	55141.286
84754	ITA	Europe	Italy	2022-02-19	12427773	50675	53491.857
84755	ITA	Europe	Italy	2022-02-20	12469975	42202	52042.857
84756	ITA	Europe	Italy	2022-02-21	12494459	24484	51429.714
84757	ITA	Europe	Italy	2022-02-22	12554596	60137	49874.571
84758	ITA	Europe	Italy	2022-02-23	12603758	49162	48345
				2022-			

Fig. 3.38: Last 100 days of Covid-19 data on Italy displayed in a datatable widget

- **show_week** (*bool, optional*) – If True the popup window will also show number of the week (default is False)
- **onchange** (*function, optional*) – Python function to call when the user selects a date. The function will receive no parameters. (default is None)
- **offset_x** (*bool, optional*) – If True the popup window will be opened on the right of the input field (default is False)
- **offset_y** (*bool, optional*) – If True the popup window will be opened on the bottom of the input field (default is True)
- **disabled** (*bool, optional*) – True if the selection of the date is disabled, False otherwise (default is False)
- **mindate** (*str, optional*) – Minimum selectable date (default is None)
- **maxdate** (*str, optional*) – Maximum selectable date (default is None)

Example

Creation of a date picker widget:

```
from vois.vuetify import datePicker
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange():
    with output:
        print('Changed to', d.date)

d = datePicker.datePicker(date=None, label='Start date',
                          offset_x=True, offset_y=False,
                          onchange=onchange)

display(d.draw())
display(output)
```

property date

Get/Set the selected date.

Returns

d – date currently selected in the format 'YYYY-MM-DD'

Return type

str

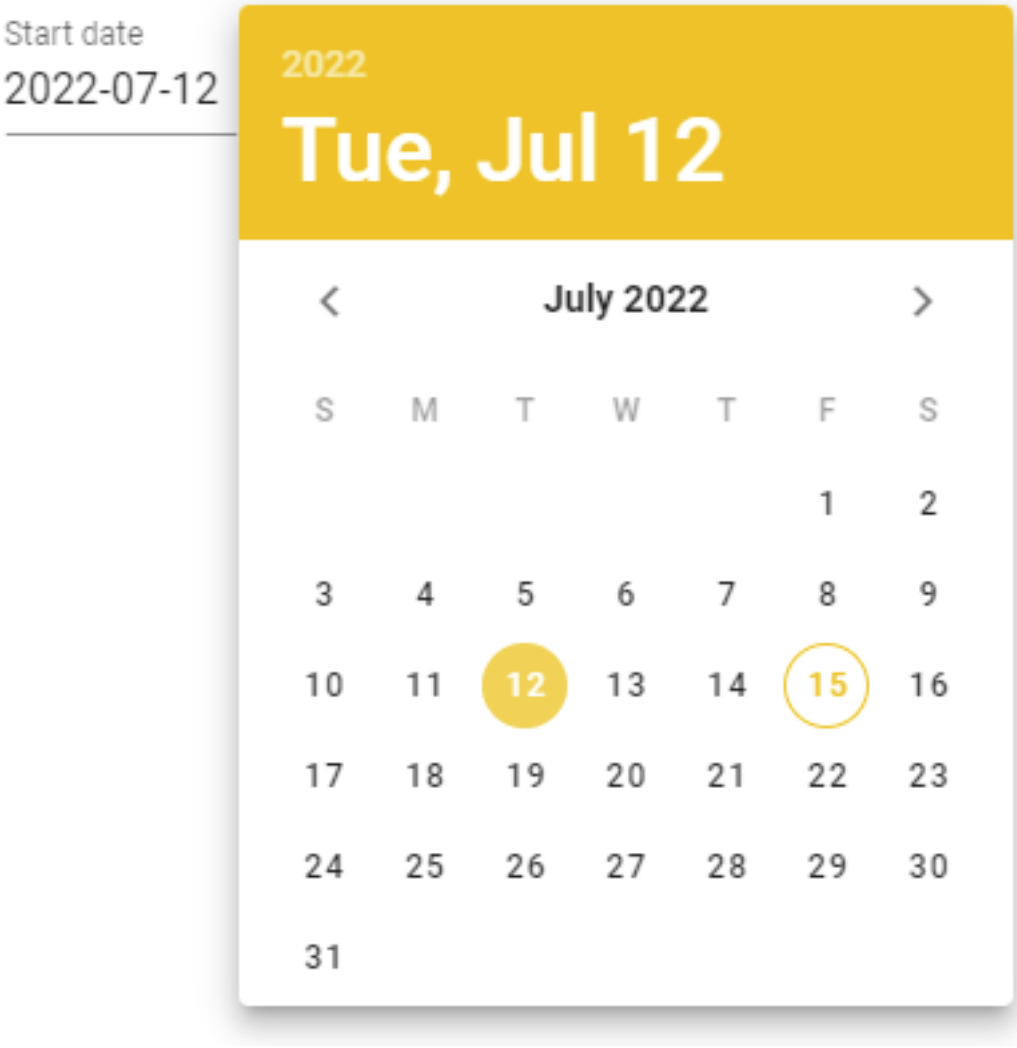


Fig. 3.39: Example of a datePicker

Example

Programmatically change the date:

```
picker.date = '2022-07-15'
print(picker.date)
```

property disabled

Get/Set the disabled state of the widget.

Returns
flag – True if the widget is disabled, False otherwise

Return type
bool

Example

Programmatically change the date:

```
picker.disabled = True
print(picker.disabled)
```

draw()

Returns the ipyvuetify object to display (the internal v.Menu)

3.3.2.9 dayCalendar widget

Calendar widget showing days with events

class dayCalendar.**dayCalendar**(*start=datetime.date(2024, 3, 20)*, *end=datetime.date(2024, 4, 20)*, *color='#f8bd1a'*, *dark=False*, *days=[]*, *show_count=False*, *width=340*, *height=None*, *on_click=None*, *on_click_event=None*)

Input widget to display a daily calendar for a range of dates and allows for highlighting some of the days and manages the click on the days.

Parameters

- **start** (*str or datetime or date instance, optional*) – Initial date of the calendar as a string in format 'YYYY-MM-DD' or as an instance of datetime.datetime or datetime.date (default is today date minus one month)
- **end** (*str or datetime or date instance, optional*) – Final date of the calendar as a string in format 'YYYY-MM-DD' or as an instance of datetime.datetime or datetime.date (default is today)
- **color** (*str, optional*) – Color to use for the highlighting of days in the calendar (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the calendar will have a dark background (default is settings.dark_mode)
- **days** (*list of str, optional*) – List of days to be highlighted as strings in "YYYY-MM-DD" format (default is []). The list can contain repeated days (see show_count below).

- **show_count** (*bool, optional*) – If True, the event bar will show the number of events on each of the highlighted days (default is False)
- **width** (*int, optional*) – Width of the widget in pixels (default is 340)
- **height** (*int, optional*) – Height of the widget in pixels (default is None). If None is passed, the height will be calculated depending on the range of dates defined by start and end parameters.
- **on_click** (*function, optional*) – Python function to call when the user clicks on one day of the calendar. The function will receive as parameter a string in "YYYY-MM-DD" format. (default is None)
- **_event** (*on_click*) – Python function to call when the user clicks on the highlighting bar of one day of the calendar. The function will receive as parameter a string in "YYYY-MM-DD" format. (default is None)

Example

Creation of a date picker widget:

```
from vois.vuetify import dayCalendar
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def on_click(day):
    with output:
        print('Clicked on ', day)

c = dayCalendar.dayCalendar(start='2023-10-01', end='2023-10-31',
                            days=['2023-10-10', '2023-10-20'],
                            on_click=on_click)

display(c.draw())
display(output)
```

property color

Get/Set the color of the highlighted days

Returns

color – Color of the highlighted days in the calendar

Return type

str

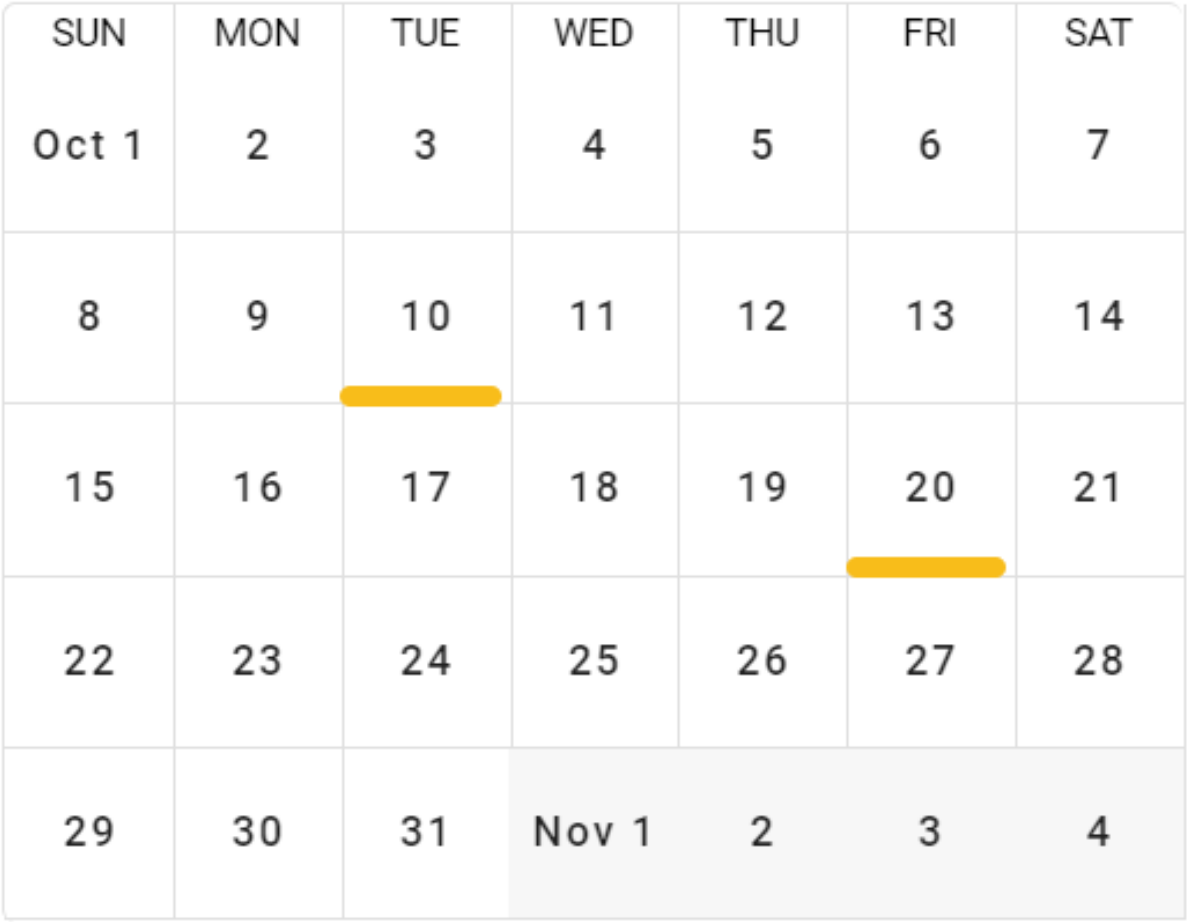


Fig. 3.40: Example of a dayCalendar

Example

Programmatically change the color:

```
cal.color = 'red'
print(cal.color)
```

property days

Get/Set the highlighted days

Returns
listfodays – List of days currently highlighted in the calendar

Return type
list of strings in "YYYY-MM-DD" format

Example

Programmatically change the highlighted days:

```
cal.days = ['2023-10-15', '2023-10-25']
print(cal.days)
```

draw()

Returns the ipyvuetify object to display (the internal Output widget)

3.3.2.10 dialogGeneric widget

Generic modal dialog-box to ask input from the user.

class dialogGeneric.**dialogGeneric**(*title='', text='', color='#f8bd1a', dark=False, show=False, content=[], width=500, fullscreen=False, persistent=False, no_click_animation=False, addclosebuttons=True, addokcancelbuttons=False, on_ok=None, on_cancel=None, on_close=None, transition='dialog-fade-transition', output=None, titleheight='dense', customclass='', custom_icon='', custom_tooltip='', custom_icon_onclick=None*)

Generic modal dialog-box to ask input from the user.

Parameters

- **title** (*str, optional*) – Title of the dialog-box to be displayed in the top toolbar (default is '')
- **text** (*str, optional*) – Text to display on top of the dialog-box body (default is '')
- **color** (*str, optional*) – Color of the title bar of the dialog (default is settings.color_first)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **show** (*bool, optional*) – Flag to immediately show the dialog-box upon creation (default is False)
- **content** (*list of ipyvuetify widgets, optional*) – List of ipyvuietify widgets to be displayed in the body of the dialog-box (default is [])

- **width** (*int or str, optional*) – Width of the dialog-box. If an integer is passed the width is intended in pixels. Default is 500 pixels
- **fullscreen** (*bool, optional*) – If True, the dialog-box is opened in fullscreen mode (default is False)
- **persistent** (*bool, optional*) – If True, clicking outside of the dialog or pressing esc key will not deactivate it (default is False)
- **no_click_animation** (*bool, optional*) – If True, disables the bounce effect when clicking outside of a dialog's content when using the persistent property (default is False)
- **addclosebuttons** (*bool, optional*) – If True, the dialog will have a 'close' button in the top toolbar (default is True)
- **addokcancelbuttons** (*bool, optional*) – If True, the dialog will have 'ok' and 'cancel' buttons in the bottom row (default is False)
- **on_ok** (*function, optional*) – Python function to call when the user clicks on the OK button. The function will receive no parameters (default is None)
- **on_cancel** (*function, optional*) – Python function to call when the user clicks on the CANCEL button. The function will receive no parameters (default is None)
- **on_close** (*function, optional*) – Python function to call when the user clicks on the CLOSE button. The function will receive no parameters (default is None)
- **transition** (*str, optional*) – Transition to use for the dialog display and close (default is 'dialog-fade-transition'. See: https://vuetifyjs.com/en/styles/transitions/ for a list of available transitions (substitute 'v-' with 'dialog-'')
- **output** (*ipywidgets.Output, optional*) – Output widget on which the widget has to be displayed
- **titleheight** (*str, optional*) – Height of the title toolbar. It can be: 'prominent', 'dense', 'extended' or a value in pixels (default is 'dense')
- **custom_icon** (*str, optional*) – Name of the optional icon to display in the top toolbar (default is '')
- **custom_tooltip** (*str, optional*) – Tooltip to display when hovering on the custom icon in the top toolbar (default is '')
- **custom_icon_onclick** (*function, optional*) – Python function to call when the user clicks on the custom icon on the topbar. The function will receive no parameters (default is None)

Example

Creation and display of a modal dialog-box containing a switch widget:

```
from vois.vuetify import dialogGeneric, switch
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def on_s_change(value):
    with output:
```

(continues on next page)

(continued from previous page)

```
        print(value)

s = switch.switch(True, 'PNG format', onchange=on_s_change)

dlg = dialogGeneric.dialogGeneric(title='Settings',
                                  text='Please select the format for download:',
                                  show=True, addclosebuttons=True, width=600,
                                  fullscreen=False, content=[s.draw()],␣
↪output=output)
```

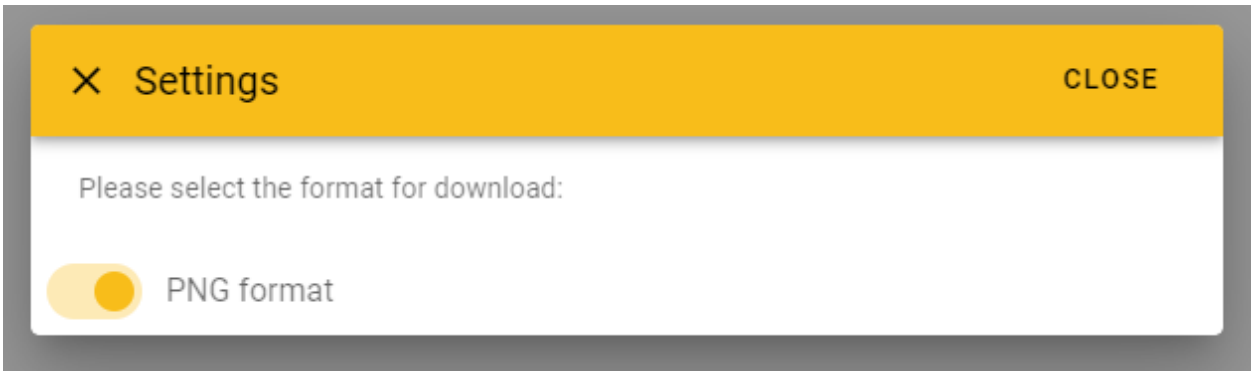


Fig. 3.41: Example of a dialogGeneric containing a switch widget.

close(**args*)

Close the dialog.

property okdisabled

Get/Set the disabled status of the ok button.

Returns

flag – True if the ok button is disabled, False otherwise

Return type

bool

Example

Programmatically set the disabled status and print it:

```
dlg.okdisabled = True
print(dlg.okdisabled)
```

show()

Open the dialog.

3.3.2.11 dialogMessage widget

Dialog-box to display a message for the user.

class dialogMessage.**dialogMessage**(**args*, ***kwargs*)

Dialog-box to display a message for the user.

Derived class from dialogGeneric.dialogGeneric.

Parameters

- **title** (*str, optional*) – Title of the dialog-box to be displayed in the top toolbar (default is '')
- **text** (*str, optional*) – Text to display on top of the dialog-box body (default is '')
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **show** (*bool, optional*) – Flag to immediately show the dialog-box upon creation (default is False)
- **width** (*int or str, optional*) – Width of the dialog-box. If an integer is passed the width is intended in pixels. Default is 500 pixels
- **addclosebuttons** (*bool, optional*) – If True, the dialog will have a 'close' button in the top toolbar (default is True)
- **transition** (*str, optional*) – Transition to use for the dialog display and close (default is 'dialog-fade-transition'). See: https://vuetifyjs.com/en/styles/transitions/ for a list of available transitions (substitute 'v-' with 'dialog-')
- **output** (*ipywidgets.Output, optional*) – Output widget on which the widget has to be displayed
- **titleheight** (*str, optional*) – Height of the title toolbar. It can be: 'prominent', 'dense', 'extended' or a value in pixels (default is 'dense')

Example

Creation and display of a modal dialog-box containing an error message:

```
from vois.vuetify import dialogMessage
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

e = dialogMessage.dialogMessage(title='Error',
                                text='''Sorry but the task could not be completed
↪<br>
because there are errors in the code to save in PNG format''',
                                addclosebuttons=False,
                                show=True, width=450, output=output)
```

close(**args*)

Close the dialog.

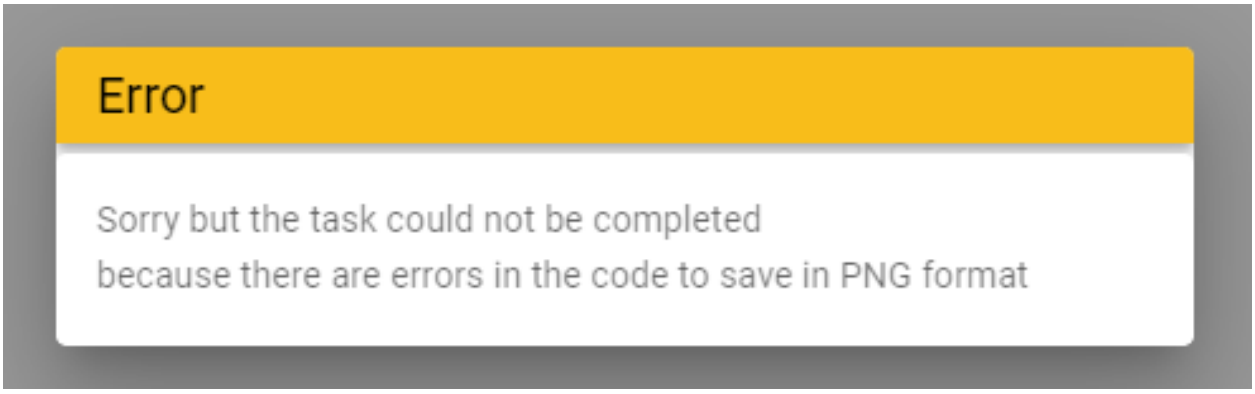


Fig. 3.42: Example of a dialogMessage to display an error message to the user.

property okdisabled

Get/Set the disabled status of the ok button.

Returns

flag – True if the ok button is disabled, False otherwise

Return type

bool

Example

Programmatically set the disabled status and print it:

```
dlg.okdisabled = True
print(dlg.okdisabled)
```

show()

Open the dialog.

3.3.2.12 dialogWait widget

Dialog-box to display a message to the user during a lenghty operation.

class dialogWait.**dialogWait**(***kwargs: Any*)

Dialog-box to display a message to the user during a lenghty operation.

Parameters

- **text** (*str, optional*) – Text to display on top of the dialog-box body (default is ‘’)
- **indeterminate** (*bool, optional*) – If True the progress bar will constantly animate (to be used when completion progress is unknown). Default is True. If set to False, setting the value property of the dialog (e.g.: dlg.value = 30) to the percentage will change the progress bar.
- **height** (*int, optional*) – Height of the progress bar in pixels (default is 4)
- **linecolor** (*str, optional*) – Color of the progress bar (default is None, meaning that the progress bar will be white or black depending on settings.dark_mode)

- **showtext** (*bool, optional*) – If True the percentage text will be displayed at the center of the progress bar (default is False)
- **textcolor** (*str, optional*) – Color of the percentage text displayed if the showtext parameter is True (default is '#cccccc')
- **output** (*ipywidgets.Output, optional*) – Output widget on which the widget has to be displayed

Example

Creation and display of dialogWait during a lenghty operation:

```
from vois.vuetify import dialogWait
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

dlg = dialogWait.dialogWait(text='Please wait for processing to terminate...',
                            output=output)
```

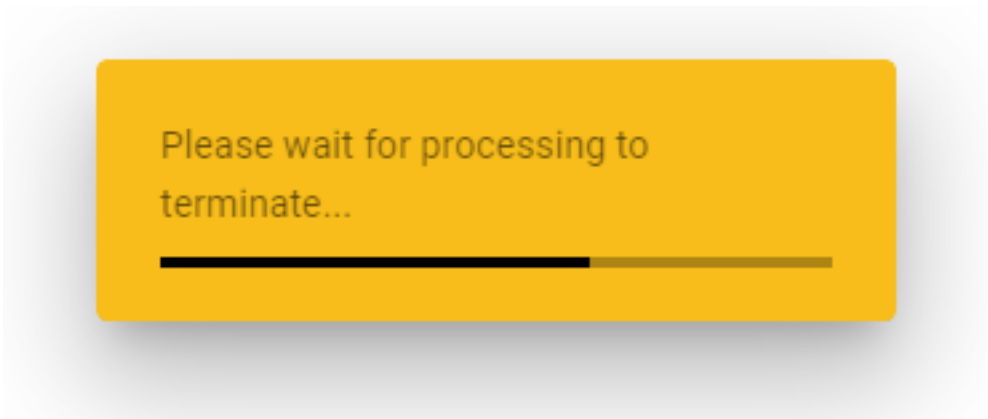


Fig. 3.43: Example of a dialogWait opened during a lenghty operation.

close()

Close the dialogWait.

3.3.2.13 dialogYesNo widget

Dialog-box to ask a yes-no question to the user.

class dialogYesNo.**dialogYesNo**(*on_yes=None*, *on_no=None*, **args*, ***kwargs*)

Dialog-box to ask a yes-no question to the user.

Derived class from dialogGeneric.dialogGeneric.

Parameters

- **title** (*str, optional*) – Title of the dialog-box to be displayed in the top toolbar (default is '')
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **show** (*bool, optional*) – Flag to immediately show the dialog-box upon creation (default is False)
- **width** (*int or str, optional*) – Width of the dialog-box. If an integer is passed the width is intended in pixels. Default is 500 pixels
- **addclosebuttons** (*bool, optional*) – If True, the dialog will have a 'close' button in the top toolbar (default is True)
- **transition** (*str, optional*) – Transition to use for the dialog display and close (default is 'dialog-fade-transition'. See: https://vuetifyjs.com/en/styles/transitions/ for a list of available transitions (substitute 'v-' with 'dialog-'')
- **output** (*ipywidgets.Output, optional*) – Output widget on which the widget has to be displayed
- **titleheight** (*str, optional*) – Height of the title toolbar. It can be: 'prominent', 'dense', 'extended' or a value in pixels (default is 'dense')
- **on_yes** (*function, optional*) – Python function to call when the user clicks on the YES button. The function will receive no parameters (default is None)
- **on_no** (*function, optional*) – Python function to call when the user clicks on the NO button. The function will receive no parameters (default is None)

Example

Creation and display of a Yes-No dialog box:

```
from vois.vuetify import dialogYesNo
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def on_yes():
    with output:
        print('YES')

def on_no():
    with output:
        print('NO')

dlg = dialogYesNo.dialogYesNo(title='Question',
                              text='Confirm removal of the selected file?',
                              titleheight=40, width=400, output=output,
                              show=True, transition='dialog-bottom-transition',
                              on_yes=on_yes, on_no=on_no)
```

close(**args*)

Close the dialog.

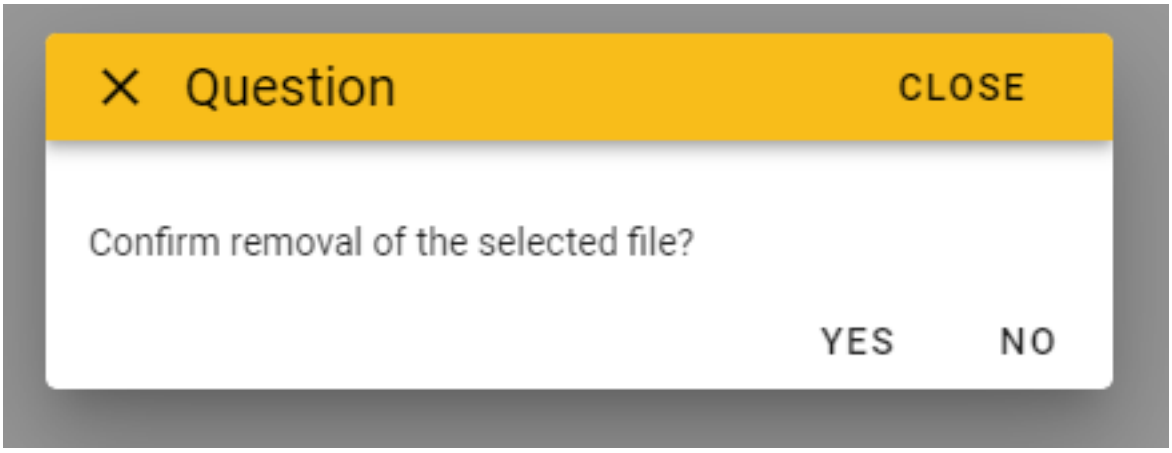


Fig. 3.44: Example of a dialogYesNo to ask a yes-no question to the user.

property okdisabled

Get/Set the disabled status of the ok button.

Returns

flag – True if the ok button is disabled, False otherwise

Return type

bool

Example

Programmatically set the disabled status and print it:

```
dlg.okdisabled = True
print(dlg.okdisabled)
```

show()

Open the dialog.

3.3.2.14 fab widget

Floating-action-button to be displayed in absolute mode on the page. It will display a menu when hovered.

class fab.**fab**(*left=100*, *top=100*, *items=[]*, *onclick=[]*, *icon='mdi-arrow-down-thick'*, *tooltipitems=[]*, *iconsmall=True*, *menumode=True*, *width=200*, *height=36*, *textcolor=None*, *selected=False*, *disabled=False*, *dark=False*, *large=False*, *small=False*, *xsmall=False*, *outlined=False*, *textweight=500*, *zindex=9999*, *output=None*)

Floating-action-button to be displayed in absolute mode on the page. It will display a menu when hovered.

Parameters

- **left** (*int or string, optional*) – Absolute left position of the button (example: '800px' or '90%' or 750)
- **top** (*int or string, optional*) – Absolute top position of the button (example: '100px' or '10%' or 120)

- **items** (*list of strings, optional*) – Strings to be displayed as text of the options (default is [])
- **onclick** (*list of function, optional*) – Python functions to call when the user clicks on one of the items. One function for each of the items (default is [])
- **tooltipitems** (*list of strings, optional*) – Tooltip text for the items (default is [])
- **iconsmall** (*bool, optional*) – Flag to display the icon in small dimension (default is True)
- **menumode** (*bool, optional*) – Flag to display the options as dropdown menu. If False, the items are displayed as horizontally displaced toggle buttons (default is True)
- **width** (*int, optional*) – Width in pixel of the toggle buttons when menumode is False (default is 200)
- **height** (*int, optional*) – Height in pixel of the toggle buttons when menumode is False (default is 36)
- **textcolor** (*str, optional*) – Color to be used for the text of the buttons when menumode is False (default is None)
- **selected** (*bool, optional*) – If True the buttons have the settings.color_first as background (default is False)
- **disabled** (*bool, optional*) – If True the buttons are disabled (default is False)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **large** (*bool, optional*) – Flag that displays the fab button larger
- **small** (*bool, optional*) – Flag that displays the fab button smaller
- **xsmall** (*bool, optional*) – Flag that displays the fab button extra smaller
- **outlined** (*bool, optional*) – Flag that displays the fab button outlined
- **textweight** (*int, optional*) – Weight of the text to be shown in the label (default is 500, Bold is any value greater or equal to 500)
- **zindex** (*int, optional*) – Z-index of the fab button (default is 9999)
- **output** (*ipywidgets.Output, optional*) – Output widget on which the widget has to be displayed

Example

Creation and display of two fab widgets for the selection among 3 options:

```
from vois.vuetify import fab
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def on1():
    with output:
        print('selected 1')
```

(continues on next page)

(continued from previous page)

```
def on2():
    with output:
        print('selected 2')

def on3():
    with output:
        print('selected 3')

b1 = fab.fab(left='70%', top='150px',
             items=['Option 1', 'Option 2', 'Option 3'],
             tooltipitems=['Tooltip for Option 1'],
             onclick=[on1,on2,on3], menumode=True,
             width=180, selected=True,
             dark=True, zindex=100, output=output)

b2 = fab.fab(left='70%', top='250px',
             items=['Option 1', 'Option 2', 'Option 3'],
             tooltipitems=['Tooltip for Option 1'],
             onclick=[on1,on2,on3], menumode=False,
             width=180, selected=True,
             dark=True, zindex=100, output=output)

b1.seticon('mdi-arrow-left-bold')
b2.seticon('mdi-arrow-left',1)
```

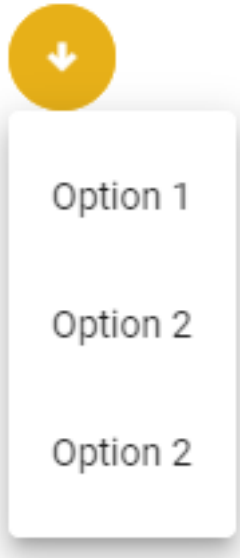


Fig. 3.45: Fab widget for selecting alternative options using a menu.

close()

Close/hide the fab button

seticon(*iconname*, *index=0*)

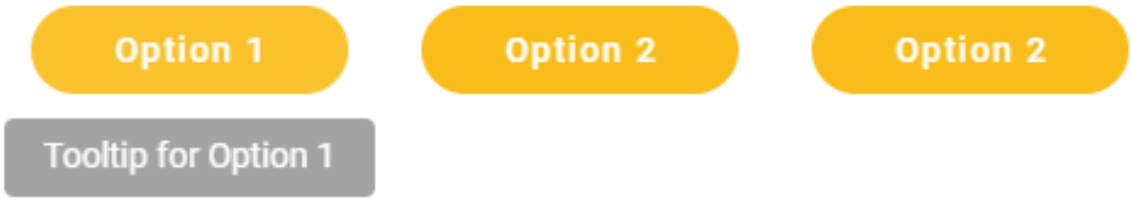


Fig. 3.46: Fab widget for selecting alternative options using toggle buttons.

Set a different icon for the fab button

settooltip(*text*, *index=0*)

Change the tooltip text of one of the buttons

show(*flag=True*)

Show/hide the fab button

3.3.2.15 footer widget

Footer bar to be displayed at the bottom of a Voilà dashboard.

class footer.**footer**(*text=''*, *copyright=False*, *buttons=[]*, *dark=False*, *height=68*, *marginy=2*, *color='#f8bd1a'*, *minipanelicons=[]*, *minipaneltooltips=[]*, *minipanelopen=False*, *minipanellarge=True*, *minipanelbuttontooltip='Additional functions'*, *minipanelbuttoncolor='#f8bd1a'*, *minipaneliconscolor='black'*, *footercredits=''*, *footercreditstooltip=''*, *onclickcredits=None*, *onclickminipanel=None*, *onclick=None*, *output=None*)

Footer bar to be displayed at the bottom of a Voilà dashboard.

Parameters

- **text** (*str, optional*) – Text message to display in the bottom line of the footer bar (default is ‘’)
- **copyright** (*bool, optional*) – Flag to display a copyright symbol in the bottom line of the footer bar (default is False)
- **buttons** (*list of strings, optional*) – List of strings to use for adding a row of buttons (default is []])
- **color** (*str, optional*) – Background color of the title bar (default is the main color defined in the settings.py module)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground: if True, the text will be displayed in white, elsewhere in black (default is the value of settings.dark_mode)
- **height** (*int or float or str optional*) – Height of the footer bar. If an integer or a float is passed, the height is intended in pixels units, otherwise a string containing the units must be passed (example: ‘4vh’). Default is 68 for 68 pixels
- **marginy** (*int, optional*) – Vertical displace of the first row of the bar from the top (default is 2)

- **onclick** (*function, optional*) – Python function to call when the user clicks on one of the buttons. The function will receive a parameter of type str containing the text of the clicked button
- **output** (*ipywidgets.Output, optional*) – Output widget on which the footer bar has to be displayed
- **minipanelicons** (*list of strings, optional*) – Names of optional icons to display in the minipanel at the left side of the footer bar
- **minipaneltooltips** (*list of strings, optional*) – Tooltips of optional icons to display in the minipanel at the left side of the footer bar
- **minipanelopen** (*bool, optional*) – If True the minipanel is initially displayed already opened (default is False)
- **minipanellarge** (*bool, optional*) – If True the icons in theminipanel are displayed with greater dimension (default is True)
- **minipanelbuttontooltip** (*str, optional*) – Text of the tooltip to show when hover on the minipanel open icon (default is 'Additional functions')
- **minipanelbuttoncolor** (*str, optional*) – Color of the icon that opens/closes the minipanel (default is the textcolor_notdark defined in the settings.py module)
- **minipaneliconscolor** (*str, optional*) – Color of the icons inside the minipanel (default is the textcolor_notdark defined in the settings.py module)
- **footercredits** (*str, optional*) – Text for footer credits button (default is '')
- **footercreditstooltip** (*str, optional*) – Tooltip for the footer credits button (default is '')
- **onclickcredits** (*function, optional*) – Python function to call when the user clicks on the credits button. The function will be called with 0 parameters
- **onclickminipanel** (*function, optional*) – Python function to call when the user clicks on one of the icons of the minipanel. The function will receive a parameter of type int containing the index of the clicked icon in the range from 0 to len(minipanelicons)-1

Example

Creation and display of a footer bar:

```
from vois.vuetify import footer
from ipywidgets import widgets
from IPython.display import display
from datetime import datetime

output = widgets.Output()
display(output)

def onclick(arg):
    with output:
        print(arg)

def onclickminipanel(index):
    with output:
        print(index)
```

(continues on next page)

(continued from previous page)

```
def onclickcredits():
    with output:
        print('CREDITS')

f = footer.footer(text='%d - Joint Research Centre'%(datetime.now().year), color=
↪'lightgrey',
                  minipanelicons=['fa-truck', 'mdi-heart', 'mdi-magnify'],
                  minipaneltooltips=['Function 1', 'Function 2', 'Function 3'],
                  minipanellarge=True, minipanelopen=True,
                  onclickminipanel=onclickminipanel,
                  footercredits='Data credits',
                  footercreditstooltip='Eurostat - European Commission',
                  onclickcredits=onclickcredits,
                  buttons=['Home', 'About Us', 'Services', 'Contact Us'],
                  height=68, marginy=2,
                  onclick=onclick,
                  output=output)
```

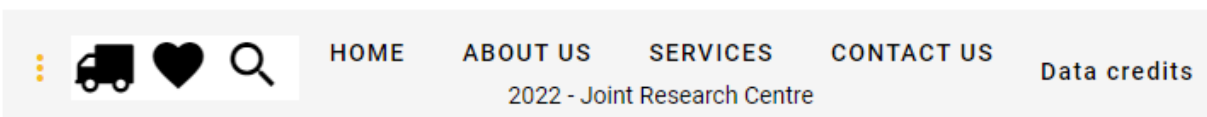


Fig. 3.47: Footer bar with buttons and minipanel.

Note: The footer component is used inside the *app.app* class. If the main interface of a dashboard is created using an instance of the *app.app* class, the parameters for customising the footer bar can be passed in the construction of the **app** instance.

draw()

Returns the ipyvuetify object to display (the internal v.Html which has a v.Footer as its only child)

3.3.2.16 iconButton widget

Button displaying an icon.

class iconButton.**iconButton**(*icon='mdi-alert-outline', tooltip='', color='#f8bd1a', outlined=False, rounded=True, width=None, large=False, small=False, x_large=False, x_small=False, margins='pa-0 ma-0', onclick=None, argument=None, disabled=False*)

Button displaying an icon.

Parameters

- **icon** (*str, optional*) – Icon to display inside the button (default is 'mdi-alert-outline')
- **tooltip** (*str, optional*) – Tooltip text for the button (default is '')

- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **outlined** (*bool, optional*) – If True applies a thin border to the button (default is False)
- **rounded** (*bool, optional*) – If True the shape of the button is rounded (default is True)
- **large** (*bool, optional*) – If True makes the button large (default is False)
- **small** (*bool, optional*) – If True makes the button small (default is False)
- **x_large** (*bool, optional*) – If True makes the button extra-large (default is False)
- **x_small** (*bool, optional*) – If True makes the button extra-small (default is False)
- **margins** (*str, optional*) – Marging apply to the button (default is 'pa-0 ma-0')
- **onclick** (*function, optional*) – Python function to call when the user clicks on the button. The function will receive no parameters
- **argument** (*any, optional*) – Argument to be passed to the onclick funtion (default is None)
- **disabled** (*bool, optional*) – If True the button will be disabled (default is False)

Example

Creation and display of an icon button which changes color and tooltip on click:

```
from vois.vuetify import iconButton
from IPython.display import display

def onclick():
    if b.color == 'red':
        b.color = 'amber'
        b.tooltip = 'Click to make the icon red'
    else:
        b.color = 'red'
        b.tooltip = 'Click to make the icon yellow'

b = iconButton.iconButton(onclick=onclick, tooltip='Initial tooltip', x_large=True)
display(b.draw())
```

property color

Get/Set the color of the button.

property disabled

Get/Set the disabled state of the button.

property tooltip

Get/Set the tooltip of the button.

3.3.2.17 label widget

Label widget to display a text with an optional icon.

class label.**label**(*text*, *onclick=None*, *argument=None*, *disabled=False*, *textweight=350*, *height=20*, *margins=0*, *margintop=None*, *icon=None*, *iconlarge=False*, *iconsmall=False*, *iconleft=False*, *iconcolor='black'*, *tooltip=None*, *textcolor=None*, *backcolor=None*, *dark=False*)

Label widget to display a text with an optional icon.

Parameters

- **text** (*str*) – Test string to be displayed on the label widget
- **onclick** (*function, optional*) – Python function to call when the user clicks on the label. The function will receive as parameter the value of the argument (default is None)
- **argument** (*any, optional*) – Argument to be passed to the onclick function when user click on the label (default is None)
- **disabled** (*bool, optional*) – Flag to show the label as disabled (default is False)
- **textweight** (*int, optional*) – Weight of the text to be shown in the label (default is 350, Bold is any value greater or equal to 500)
- **height** (*int, optional*) – Height of the label widget in pixels (default is 20)
- **margins** (*int, optional*) – Dimension of the margins on all directions (default is 0)
- **margintop** (*int, optional*) – Dimension of the margin on top of the label (default is None)
- **icon** (*str, optional*) – Name of the icon to display aside the text of the label (default is None)
- **iconlarge** (*bool, optional*) – Flag that sets the large version of the icon (default is False)
- **iconsmall** (*bool, optional*) – Flag that sets the small version of the icon (default is False)
- **iconleft** (*bool, optional*) – Flag that sets the position of the icon to the left of the text of the label (default is False)
- **iconcolor** (*str, optional*) – Color of the icon (default is 'black')
- **tooltip** (*str, optional*) – Tooltip string to display when the user hovers on the label (default is None)
- **textcolor** (*str, optional*) – Color used for the label text
- **backcolor** (*str, optional*) – Color used for the background of the label
- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)

Note: All the icons from https://materialdesignicons.com/ site can be used, just by prepending 'mdi-' to their name.

All the free icons from https://fontawesome.com/ site can be used, just by prepending 'fa-' to their name.

Example

Creation and display of a label widget containing an icon:

```
from vois.vuetify import label

lab = label.label('Test label', textweight=300, margins=2,
                  icon='mdi-car-light-high', iconcolor='red',
                  iconlarge=True, height=22)

display(lab.draw())
```

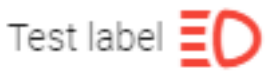


Fig. 3.48: Example of a label widget with text and an icon.

draw()

Returns the ipyvuetify object to display (the internal v.Html which has a v.Container as its only child)

property text

Get/Set the label text.

Returns

text – Text currently shown in the label

Return type

str

Example

Programmatically set the label text (needs a re-display to be visible!):

```
lab.text = 'New text for the label'
display(lab.draw())
print(lab.text)
```

3.3.2.18 layers widget

Widget to manage the layers Table Of Content for a ipyleaflet Map

class layers.**interaproLayer**(*name*, *visible=True*, *opacity=1.0*, *path=None*, *file=None*, *epsg=None*, *nodata=None*, *colorfile=None*, *colortable=None*, *colormap=None*, *valuemap=None*, *colorscheme=None*, *colorcustom=None*, *scalemin=None*, *scalemax=None*, *interpolate=None*, *bands=None*, *rmin=None*, *rmax=None*, *gmin=None*, *gmax=None*, *bmin=None*, *bmax=None*)

A Layer to be visualized by interapro

edit(*output*, *outdebug*, *onok=None*, *oncancel=None*)

Open a dialog-box to edit the layer

tileLayer()

Return a ipyleaflet.TileLayer instance

class layers.**layers**(*m*, *color='#f8bd1a'*, *dark=False*, *width=400*)

Widget to manage the layers Table Of Content for a ipyleaflet Map

Parameters

- **m** (*ipyleaflet.Map instance*) – Map instance to connect to the layers widget
- **color** (*str, optional*) – Color to use for the widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the widget will have a dark background (default is settings.dark_mode)
- **width** (*int, optional*) – Width of the widget in pixels (default is 320)

Example

Creation of a layers management widget:

```
import ipyleaflet
from jeodpp import inter, imap
from ipywidgets import widgets, Layout

from vois.vuetify import layers

# Given a collection path returns a ipyleaflet.TileLayer
def tileLayer(name, collectionpath):
    p = inter.Collection(collectionpath).process()
    procid = p.toLayer()
    return ipyleaflet.TileLayer(name=name, url='https://jeodpp.jrc.ec.europa.eu/
↪jeodpp-inter-view/?x={x}&y={y}&z={z}&procid=%s'%procid)

layer1 = tileLayer('Merit DEM',    inter.collections.BaseData.Elevation.MERIT.
↪Hillshade)
layer2 = tileLayer('Corine 2018',  inter.collections.BaseData.Landcover.CLC2018)
layer3 = tileLayer('Gisco Labels', inter.collections.Basemaps.Gisco.OSMCartoLabels)

height = 500
m = imap.Map(layout=Layout(height='%dpx'%height))

m.add_layer(layer1)
m.add_layer(layer2)
m.add_layer(layer3)

ly = layers.layers(m, width=400, dark=False)

display(widgets.HBox([ly.draw(),m]))
```

draw()

Returns the ipyvuetify object to display (the internal card containing the widgets)

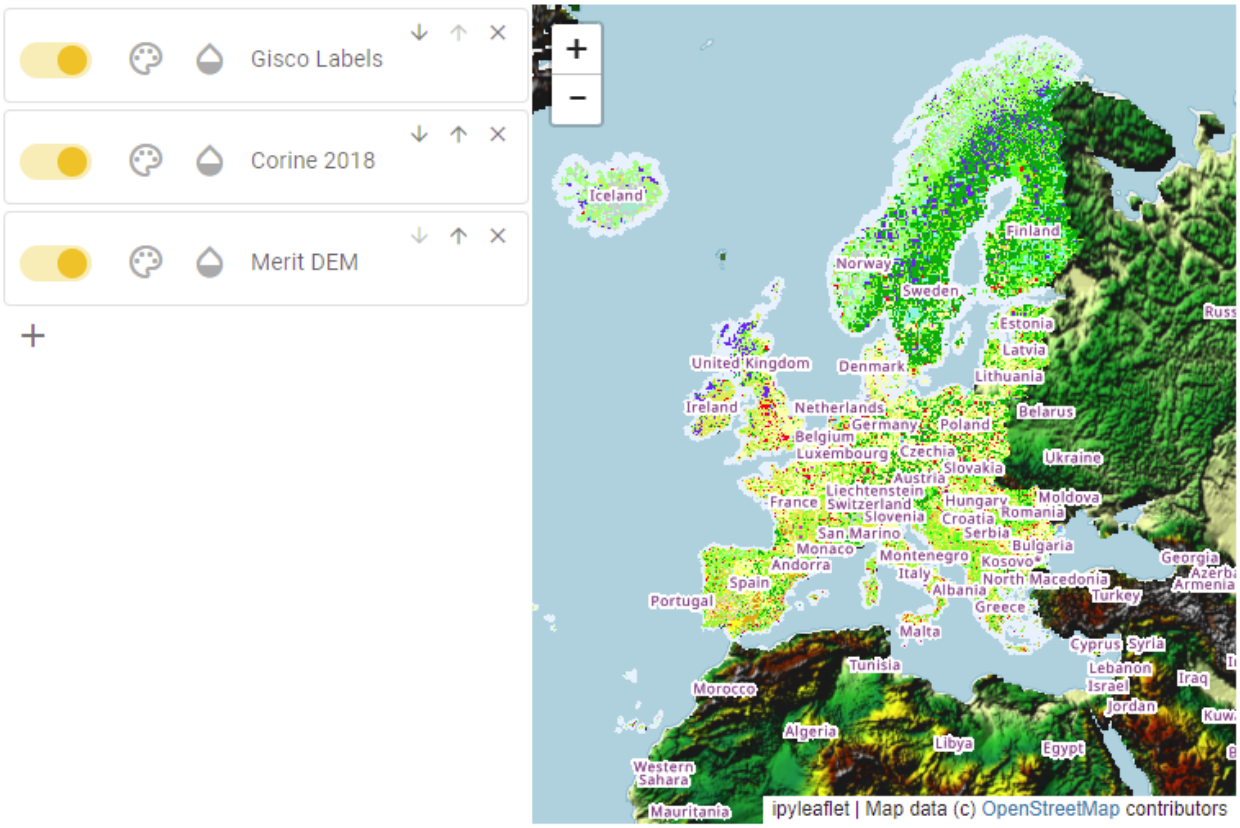


Fig. 3.49: Example of a layers management widget

3.3.2.19 mainPage widget

Initial page for an application

class mainPage.**mainPage**(*title='Application main title', subtitle='Subtitle to be shown below the main title', applogo_url='https://jeodpp.jrc.ec.europa.eu/services/shared/pngs/sampleapplogo.png', applogo_widthpercent=35.0, credits='Credits Team XXX', creditslogo_url='https://jeodpp.jrc.ec.europa.eu/services/shared/pngs/TransparentJRC.png', text_color='#0e446e', background_image=22, background_filter='', vois_show=True, vois_opacity=0.4, titlebox_widthpercent=40.0, titlebox_heightpercent=24.0, titlebox_toppercent=14.0, titlebox_opacity=0.4, titlebox_border=4, buttonbox_widthpercent=80.0, buttonbox_heightpercent=50.0, buttonbox_toppercent=43.0, creditbox_widthpercent=80.0, creditbox_heightpercent=17.0, creditbox_toppercent=80.0, creditbox_opacity=0.6, button_widthpercent=30.0, button_heightpercent=10.0, button_elevation=6, button_opacity=0.5, disclaimer=''*)

Initial page of an application. The page contains a box for the title/subtitle/logo of the application, a box containing one or more buttons, each of them calling a callback function, and a box containing credits information. All these boxes are horizontally centered inside the page, while their vertical position can be customized (by default the title is on top, the buttons in central position and the credit box is at the bottom of the page).

Parameters

- **title** (*str, optional*) – Title of the page (default is 'Application main title')
- **subtitle** (*str, optional*) – Subtitle of the page (default is 'Subtitle to be displayed below the main title')
- **applogo_url** (*str, optional*) – Url of the application logo (default is a sample logo)
- **applogo_widthpercent** (*float, optional*) – Width of the area where the application logo is displayed in percentage of the screen (default is 35.0)
- **credits** (*str, optional*) – Credits string to display inside the credit box on the bottom of the page (default is 'Credits Team XXX')
- **creditslogo_url** (*str, optional*) – Url of the image to display inside the credit box on the bottom of the page (default is 'https://jeodpp.jrc.ec.europa.eu/services/shared/pngs/TransparentJRC.png')
- **text_color** (*str, optional*) – Color to use for text (title, subtitle, credits, buttons text, etc.). Default is '#0e446e'.
- **background_image** (*str or int, optional*) – Image to use as fullscreen background. If an integer in the range [0,59] is passed, one of the sixty predefined wallpapers is used, otherwise any image URL can be used (default is 22)
- **background_filter** (*str, optional*) – CSS image filter to apply to the background image (default is ''). See https://developer.mozilla.org/en-US/docs/Web/CSS/filter for a list of available filters.
- **vois_show** (*bool, optional*) – Is True, a credit to the vois library is displayed in the top-right side of the page (default is True)
- **vois_opacity** (*float, optional*) – Opacity to apply to the vois logo, in case vois_show is set to True (default is 0.4)
- **titlebox_widthpercent** (*float, optional*) – Width of the top box containing the title, in percentage of the screen width. Default is 40.0.

- **titlebox_heightpercent** (*float, optional*) – Height of the box containing the title, in percentage of the screen height. Default is 24.0.
- **titlebox_toppercent** (*float, optional*) – Top position of the box containing the title, in percentage of the screen height, measured from the top of the page. Default is 14.0.
- **titlebox_opacity** (*float, optional*) – Opacity to apply to the box containing the title (default is 0.4)
- **titlebox_border** (*int, optional*) – Border width in pixels of the box containing the title (default is 4)
- **buttonbox_widthpercent** (*float, optional*) – Width of the box containing the buttons, in percentage of the screen width. Default is 80.0.
- **buttonbox_heightpercent** (*float, optional*) – Height of the box containing the buttons, in percentage of the screen height. Default is 50.0.
- **buttonbox_toppercent** (*float, optional*) – Top position of the box containing the buttons, in percentage of the screen height, measured from the top of the page. Default is 43.0.
- **creditbox_widthpercent** (*float, optional*) – Width of the box containing the credits, in percentage of the screen width. Default is 80.0.
- **creditbox_heightpercent** (*float, optional*) – Height of the box containing the credits, in percentage of the screen height. Default is 17.0.
- **creditbox_toppercent** (*float, optional*) – Top position of the box containing the credits, in percentage of the screen height, measured from the top of the page. Default is 80.0.
- **creditbox_opacity** (*float, optional*) – Opacity to apply to the box containing the credits (default is 0.6)
- **button_widthpercent** (*float, optional*) – Width of each of the buttons, in percentage of the screen width. Default is 30.0.
- **button_heightpercent** (*float, optional*) – Height of each of the buttons, in percentage of the screen height. Default is 10.0.
- **button_elevation** (*int, optional*) – Elevation in pixels to apply to the buttons (default is 6)
- **button_opacity** (*float, optional*) – Opacity to apply to the buttons (default is 0.5)
- **disclaimer** (*str, optional*) – Text to display at the bottom of the creditbox (default is the empty string)

Examples

Creation of a main page with 6 buttons displaying random images from https://picsum.photos/:

```
from vois.vuetify import mainPage
from random import randrange

def onclick1():
    print("Clicked Function 1")

m = mainPage.mainPage(title='mainPage demo',
```

(continues on next page)

(continued from previous page)

```
                        subtitle='Showcase how easy is to create a front page for an␣
↪app',
                        credits="vois library development team",
                        titlebox_widthpercent=50, titlebox_opacity=0.2, titlebox_
↪border=0,
                        vois_show=True, vois_opacity=0.1,
                        button_widthpercent=23, button_heightpercent=14, button_
↪elevation=16, button_opacity=0.6,
                        background_image=55,
                        background_filter='blur(2px) brightness(1.2) contrast(0.7)␣
↪sepia(0.05) saturate(1.2)',
                        creditbox_opacity=0,
                        text_color='#222222')

m.addButton('Function 1',
            subtitle='Launch calculation of ...',
            tooltip='Tooltip text to display on hover',
            image='https://picsum.photos/seed/%d/200/200'%randrange(1000),
            onclick=onclick1)

for i in range(2,7): m.addButton('Function %d'%i, image='https://picsum.photos/seed/
↪%d/200/200'%randrange(1000))

m.open()
```

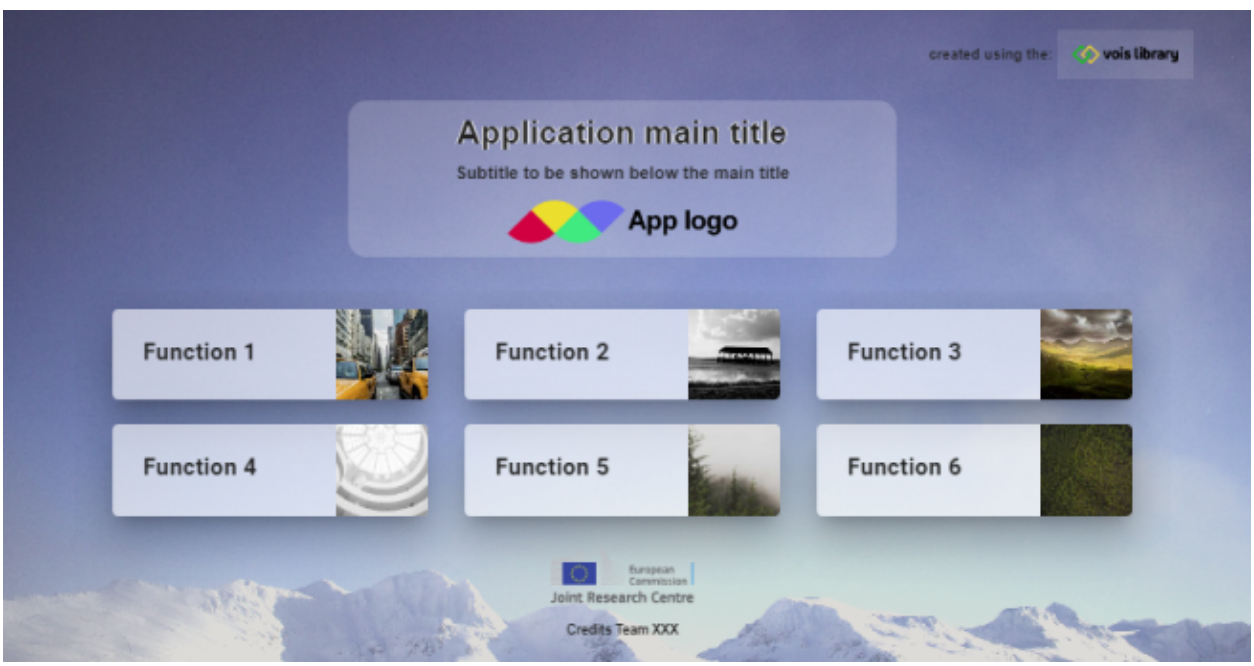


Fig. 3.50: Example of a mainPage

addButton(*title*, *subtitle=''*, *tooltip=''*, *image=''*, *onclick=None*, *argument=None*)

Add a button to the page

Parameters

- **title** (*str*) – Title of the button
- **subtitle** (*str, optional*) – Subtitle of the button (default is ‘’)
- **tooltip** (*str, optional*) – Tooltip to show when hovering the button title (default is ‘’)
- **image** (*str, optional*) – Image to show on the right side of the button (default is ‘’)
- **onclick** (*function, optional*) – Python function to call when the user clicks on the button. The function will receive the argument value as parameter if it is not None, otherwise it will be calle with no parameters. Default is None
- **argument** (*any, optional*) – Argument to pass to the onclick python function (default is None)

close()

Close the page

open()

Open the page

3.3.2.20 menu widget

Menu widget opened on hover on a button.

class menu.**menu**(*index*, *title*, *labels*, *color='#f8bd1a'*, *onchange=None*, *width=150*, *highliteselection=True*)

Menu widget opened on hover on a button.

Parameters

- **index** (*int*) – Index of the option initially selected (from 0 to len(labels)-1)
- **title** (*str*) – Title string to display in the button
- **labels** (*list of strings*) – Strings to be displayed as options in the menu widget
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the values in the list. The function will receive a single parameter, containing the index of the selected option in the range from 0 to len(labels)-1
- **width** (*int, optional*) – Width of the button in pixels (default is 150 pixels)
- **highliteselection** (*bool, optional*) – If True, the menu will show the selected option in bold (default is True)

Example

Creation and display of a menu widget to select among three options:

```
from vois.vuetify import menu
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(value):
    with output:
        print(value)

m = menu.menu(0, 'Hover to select',
              ['Option 0', 'Option 1', 'Option 2'],
              onchange=onchange, highliteselection=True)

display(m.draw())
display(output)
```

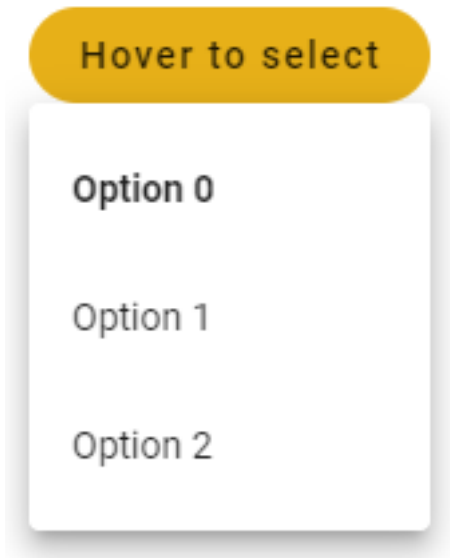


Fig. 3.51: Example of a menu widget to select from three options.

draw()

Returns the ipyvuetify object to display (the internal v.Menu widget)

property value

Get/Set the active option index.

Returns

index – Index of the selected option (from 0 to len(labels)-1)

Return type

int

Example

Programmatically select one of the options and print the index of the selected option:

```
m.value = 2
print(m.value)
```

3.3.2.21 multiSwitch widget

Widget to select independent options using a list of buttons displayed horizontally or vertically.

class multiSwitch.**multiSwitch**(*values*, *labels*, *tooltips=None*, *color='#f8bd1a'*, *onchange=None*, *dark=False*, *row=True*, *width=150*, *height=36*, *justify='space-between'*, *rounded=True*, *outlined=False*, *colorselected='#f8bd1a'*, *colorunselected='#efefef'*, *managedblclick=False*, *paddingrow=1*, *paddingcol=2*, *tile=False*, *small=False*, *xsmall=False*, *large=False*, *xlarge=False*)

Widget to select independent options using a list of buttons displayed horizontally or vertically.

Parameters

- **values** (*list of bool*) – Initial state of the buttons representing the independent options
- **labels** (*list of strings*) – Strings to be displayed as text of the options
- **tooltips** (*list of strings, optional*) – Tooltip text for the options
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)
- **onchange** (*function, optional*) – Python function to call when the user clicks on one of the buttons. The function will receive a parameter of list containing the status of all the buttons, from 0 to len(labels)-1
- **row** (*bool, optional*) – Flag to display the buttons horizontally or vertically (default is True)
- **width** (*int, optional*) – Width in pixels of the buttons (default is 150)
- **height** (*int, optional*) – Height in pixels of the buttons (default is 36)
- **justify** (*str, optional*) – In case of horizontal placement, applies the justify-content css property. Available options are: start, center, end, space-between and space-around.
- **rounded** (*bool, optional*) – Flag to display the buttons with a rounded shape (default is the button_rounded flag defined in the settings.py module)
- **outlined** (*bool, optional*) – Flag to display the buttons as outlined (default is True)
- **colorselected** (*str, optional*) – Color used for the buttons when they are selected (default is settings.color_first)
- **colorunselected** (*str, optional*) – Color used for the buttons when they are not selected (default is settings.color_second)
- **managedblclick** (*bool, optional*) – If True the dblclick event is managed to select a single button of the multi-switch (default is False)

- **paddingrow** (*int, optional*) – Horizontal padding among toggle buttons (1 unit means 4 pixels). Default is 1
- **paddingcol** (*int, optional*) – Vertical padding among toggle buttons (1 unit means 4 pixels). Default is 2
- **tile** (*bool, optional*) – Flag to remove the buttons small border (default is False)
- **large** (*bool, optional*) – Flag that sets the large version of the button (default is False)
- **xlarge** (*bool, optional*) – Flag that sets the xlarge version of the button (default is False)
- **small** (*bool, optional*) – Flag that sets the small version of the button (default is False)
- **xsmall** (*bool, optional*) – Flag that sets the xsmall version of the button (default is False)

Example

Creation and display of a widget for the selection of 3 independent options:

```
from vois.vuetify import multiSwitch
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(values):
    with output:
        print(values)

m = multiSwitch.multiSwitch([False, True, False], ['Option 1', 'Option 2', 'Option 3
↪'],
                            tooltips=['Tooltip for option 1'],
                            onchange=onchange, row=False, width=150, rounded=False,␣
↪outlined=True,
                            colorselected='#FFA300', colorunselected='#aaaaaa')

display(m.draw())
display(output)
```

Fig. 3.52: multiSwitch widget for selecting independent options using buttons.

draw()

Returns the ipyvuetify object to display (the internal v.Row or v.Col widget)

property value

Get/Set the status of the multiSwitch buttons.

Returns

flags – Selected status of all the options

Return type

list of booleans

Example

Programmatically set the options and print the status of the multiSwitch buttons:

```
t.value = [False, True, True]
print(t.value)
```

3.3.2.22 page widget

Fullscreen page

class page.**page**(*appname*, *title*, *output*, *onclose=None*, *titlecolor='#f8bd1a'*, *titledark=True*, *titleheight=54*, *footercolor='#efefef'*, *footerdark=False*, *footerheight=30*, *logoappurl=''*, *logowidth=40*, *on_logoapp=None*, *copyrighttext=''*, *show_back=True*, *show_help=True*, *on_help=None*, *logocreditsurl=''*, *show_credits=True*, *on_credits=None*, *transition='dialog-bottom-transition'*, *persistent=False*)

Fullsceen page with title and footer bar.

Parameters

- **appname** (*str*) – Name of the application. It will be displayed on the left side of the title bar
- **title** (*str*) – Title of the page
- **output** (*instance of widgets.Output() class*) – Output widget to be used for the opening of the fullscreen dialog that implements the page
- **onclose** (*function, optional*) – Python function to call when the user closes the page. The function will receive no parameters (default is None)
- **titlecolor** (*str, optional*) – Color to use for the title bar background (default is settings.color_first)
- **titledark** (*bool, optional*) – If True the text on the title bar will be displayed in white, otherwise in black color (defaul is True)
- **titleheight** (*int, optional*) – Height of the title bar in pixels (default is 54)
- **footercolor** (*str, optional*) – Color to use fir the footer bar background (default is settings.color_second)
- **footerdark** (*bool, optional*) – If True the text on the footer bar will be displayed in white, otherwise in black color (defaul is False)
- **footerheight** (*int, optional*) – Height of the footer bar in pixels (default is 30)
- **logoappurl** (*str, optional*) – String containing the url of the application logo, to be displayed on the left side of the title bar (default is ‘’)
- **logowidth** (*int, optional*) – Width in pixels of the logo button (default is 40)
- **on_logoapp** (*function, optional*) – Python function to call when the user clicks on the pplication logo. The function will receive no parameters (default is None)
- **copyrighttext** (*str, optional*) – Text to display as copyright message on the footer bar (default is ‘’)
- **show_back** (*bool, optional*) – If True a “back” button is displayed on the right side of the title bar (default is True)

- **show_help** (*bool, optional*) – If True a "help" button is displayed on the right side of the title bar (default is True)
- **on_help** (*function, optional*) – Python function to call when the user clicks the help button. The function will receive no parameters (default is None)
- **logocreditsurl** (*str, optional*) – String containing the url of the credits logo, to be displayed on the right side of the title bar (default is ''). If no url is passed, the logo of the European Commission is displayed
- **show_credits** (*bool, optional*) – If True a "credits" button is displayed on the right side of the title bar (default is True)
- **on_credits** (*function, optional*) – Python function to call when the user clicks the credits button. The function will receive no parameters (default is None)
- **transition** (*str, optional*) – Transition to be used for display and hide of the page (default is 'dialog-bottom-transition'). See: https://vuetifyjs.com/en/styles/transitions/ for a list of available transitions (substitute 'v-' with 'dialog-')
- **persistent** (*bool, optional*) – If True the page will be persistent and not close at the hitting of the "ESC" key (default is False)

Examples

Creation of an example page:

```
from vois.vuetify import page
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onclose():
    pass

def on_click():
    pass

p = page.page('Application XYZ', 'Map page', output, onclose=onclose,
              titlecolor='#008800', titledark=True,
              footercolor='#cccccc', footerdark=False,
              logoappurl='https://jeodpp.jrc.ec.europa.eu/services/shared/pngs/BDAP_
↪Logo1024transparent.png',
              on_logoapp=on_click, copyrighttext='European Commission - Joint␣
↪Research Centre',
              show_back=True, show_help=True, on_help=on_click,
              show_credits=True, on_credits=on_click)

card = p.create()
card.children = []
p.open()
```

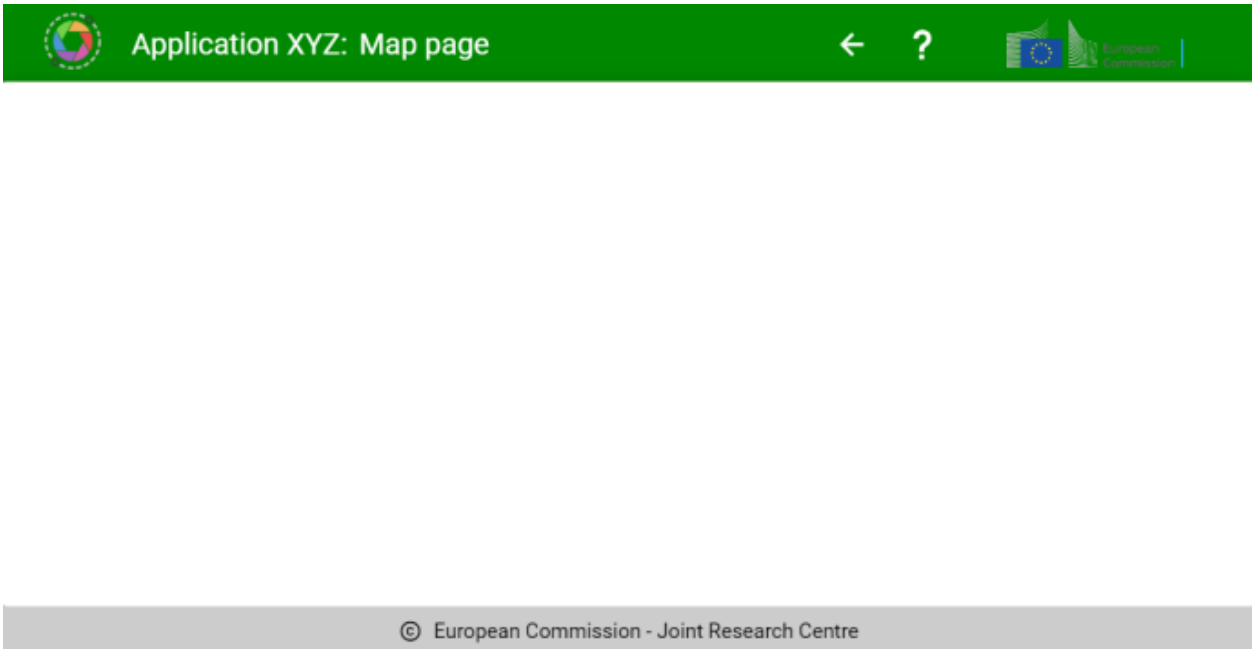


Fig. 3.53: Example of a page

3.3.2.23 paletteEditor widget

Widget for the creation and editing of color palettes.

`paletteEditor.`**`colorlist2Items`**(*colors=[]*)

Utility function to convert a list of colors to a list of items to pass as parameter to the `paletteEditor.paletteEditor` class

Parameters

colorlist (*list of str, optional*) – List of string representing colors in the format '#RRGGBB' (default is [])

Return type

A list of items ready to be passed to the `paletteEditor.paletteEditor` class constructor

class `paletteEditor.`**`paletteEditor`**(*title='', items=[], interpolate=True, width=420, maxheightlist=600, color='#f8bd1a', dark=False, onchange=None, buttonstooltip=True*)

Widget for the creation and editing of color palettes.

Parameters

- **title** (*str, optional*) – Title of the palette (default is '')
- **items** (*list of dicts, optional*) – List of dicts containing "value", "class" and "color" (default is [])
- **interpolate** (*bool, optional*) – Flag to control the interpolation of the colorlist: if True the palette will be displayed by adding intermediate colors (default is True)
- **width** (*int, optional*) – Width of the widget in pixels (default is 420)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)

- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)
- **onchange** (*function, optional*) – Python function to call when the user changes the order of colors or removes a color. The function will receive no parameters as input (default is None)
- **buttonstooltip** (*bool, optional*) – If True, the buttons to mode, add, remove colors and assign values to colors will have a tooltip (default is True)

Example

Example of a widget to create and edit a color palette:

```
from vois.vuetify import paletteEditor
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange():
    with output:
        print('onchange!')

p = paletteEditor(items=[], width=450, onchange=onchange)

display(p.draw())
display(output)
```

property colors

Get/set the colors of the palette.

Returns
colorlist – List of colors of the palette

Return type
list of strings in '#RRGGBB' format

Example

Get the edited palette colors:

```
print(editor.colors)
```

draw()

Returns the ipyvuetify object to display (a v.Html object displaying two output widgets)

property interpolate

Get/set the interpolate flag.

Returns
flag – Interpolate flag

Return type
bool

Fig. 3.54: Example of a widget to create and edit a color palette

property items

Get/set the items of the palette.

Returns
items – List of dicts containing "value", "class" and "color"

Return type
list of dicts

Example

Print the edited palette items:

```
print(editor.items)
```

property title

Get/set the title of the palette.

Returns
t – Title of the palette

Return type
str

3.3.2.24 palettePicker widget

Selection of a palette of colors

class palettePicker.**palettePicker**(*family='sequential', label='', interpolate=True, width=400, height=34, custompalettes=[], clearable=True, color='#f8bd1a', onchange=None*)

Selection of a palette of colors.

Parameters

- **family** (*str, optional*) – Family of the palette, one of these values: ['carto', 'cmocean', 'cyclical', 'diverging', 'plotlyjs', 'qualitative', 'sequential', 'custom']. If family is 'custom' the user of the widget has to pass the list of palettes to display (default is 'sequential')
- **custompalettes** (*list of dicts containing tags: "name" and "colors", optional*) – Custom palette to be selected when the family is 'custom' (default is [])
- **label** (*str, optional*) – Label to display inside the selection widget (default is '')
- **interpolate** (*bool, optional*) – If True the colors are displayed as interpolated (default is True)
- **width** (*int, optional*) – Width of the widget in pixels (default is 400)
- **height** (*int, optional*) – Height of the widget in pixels (default is 34)
- **clearable** (*Bool, optional*) – If True the selection widget will show a -x- button to clear the selection (default is True)
- **color** (*str, optional*) – Color of the selection widget (default is settings.color_first)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the palettes. The function will receive no parameters (default is None)

Examples

Creation of a simple selection widget for the palettes:

```
from vois.vuetify import palettePicker
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange():
    with output:
        print('changed!')

p = palettePicker.palettePicker(onchange=onchange)

display(p.draw())
display(output)
```

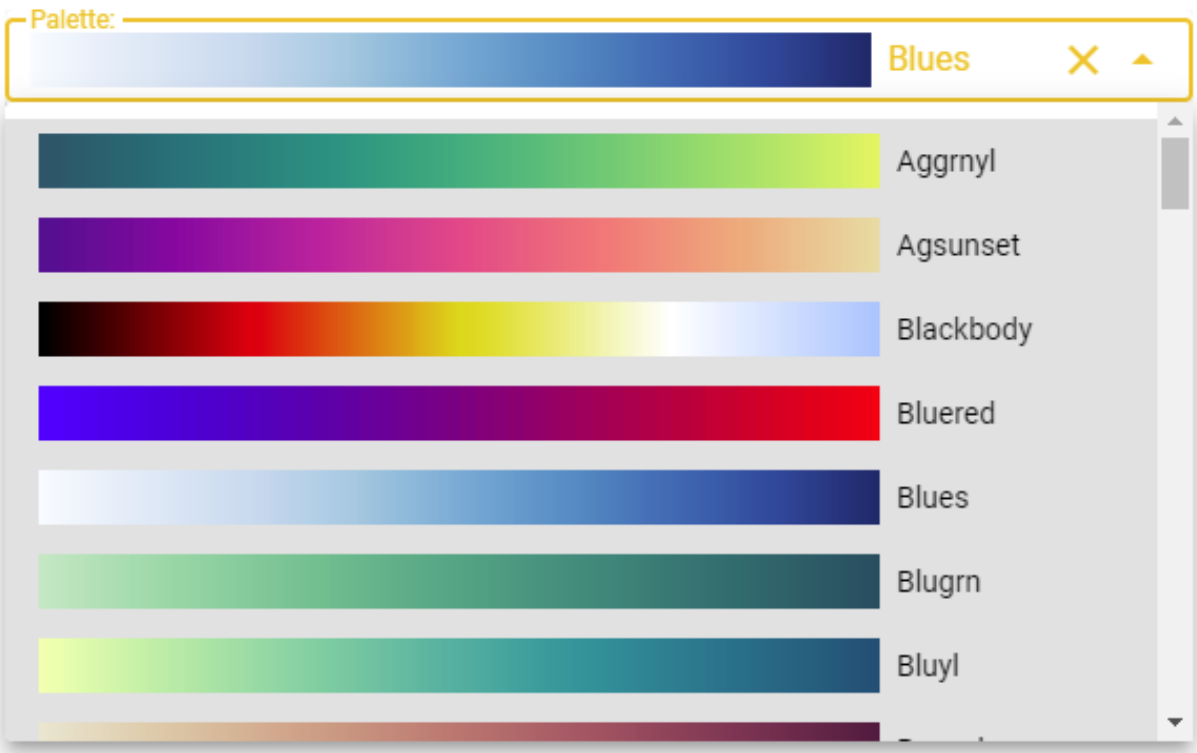


Fig. 3.55: Example of a simple palette picker

Creation of a complex selection widget for the palettes that manages all the palette families:

```
from vois.vuetify import palettePicker, selectSingle, switch
import ipyvuetify as v
from ipywidgets import widgets
from IPython.display import display
```

(continues on next page)

(continued from previous page)

```
output = widgets.Output()

# Utility: convert three integers to '#RRGGBB'
def RGB(r,g,b):
    return '#{:02X}{:02X}{:02X}'.format(r, g, b)

# Custom palettes
custompalettes = [
    { "name": "Simple",
      "colors": ['#000000', '#FF0000', '#00FF00', '#0000FF',
                 '#FFFF00', '#FF00FF', '#00FFFF', '#FFFFFF']},

    { "name": "Dem",
      "colors": [RGB(255,255,170), RGB( 39,168, 39),
                 RGB( 11,128, 64), RGB(255,255,  0),
                 RGB(255,186,  3), RGB(158, 30,  2),
                 RGB(110, 40, 10), RGB(138, 94, 66),
                 RGB(255,255,255)]},

    { "name": "NDVI",
      "colors": [RGB(120,69,25),   RGB(255,178,74),
                 RGB(255,237,166), RGB(173,232,94),
                 RGB(135,181,64),  RGB(3,156,0),
                 RGB(1,100,0),     RGB(1,80,0)]}
]

families = ['carto', 'cmocean', 'cyclical', 'diverging',
            'plotlyjs', 'qualitative', 'sequential', 'custom']

family      = 'sequential'
interpolate = True

p = None
def onchangePalette():
    if not p is None:
        with output:
            print("Palette changed to", p.value, p.colors)


def onchangeFamily():
    global family, interpolate
    family = sel.value
    if family == 'carto' or family == 'qualitative':
        interpolate = False
        sw.value = interpolate
    else:
        interpolate = True
        sw.value = interpolate
    p.updatePalettes(family,interpolate)


def onchangeInterpolate(flag):
```

(continues on next page)

(continued from previous page)

```
    global interpolate
    interpolate = flag
    p.updatePalettes(family,interpolate)


sel = selectSingle.selectSingle('Family:', families, selection=family,
                                width=160, onchange=onchangeFamily,
                                marginy=1, clearable=False)
sw  = switch.switch(interpolate, "Interpolate",
                    onchange=onchangeInterpolate)

p = palettePicker.palettePicker(family=family, custompalettes=custompalettes,
                                label='Palette:', height=26,␣
↪onchange=onchangePalette)

spacer = v.Html(tag='div',children=[' '], style_='width: 10px;')

display(widgets.HBox([sel.draw(), spacer, p.draw(), spacer, sw.draw()]))
display(output)
```

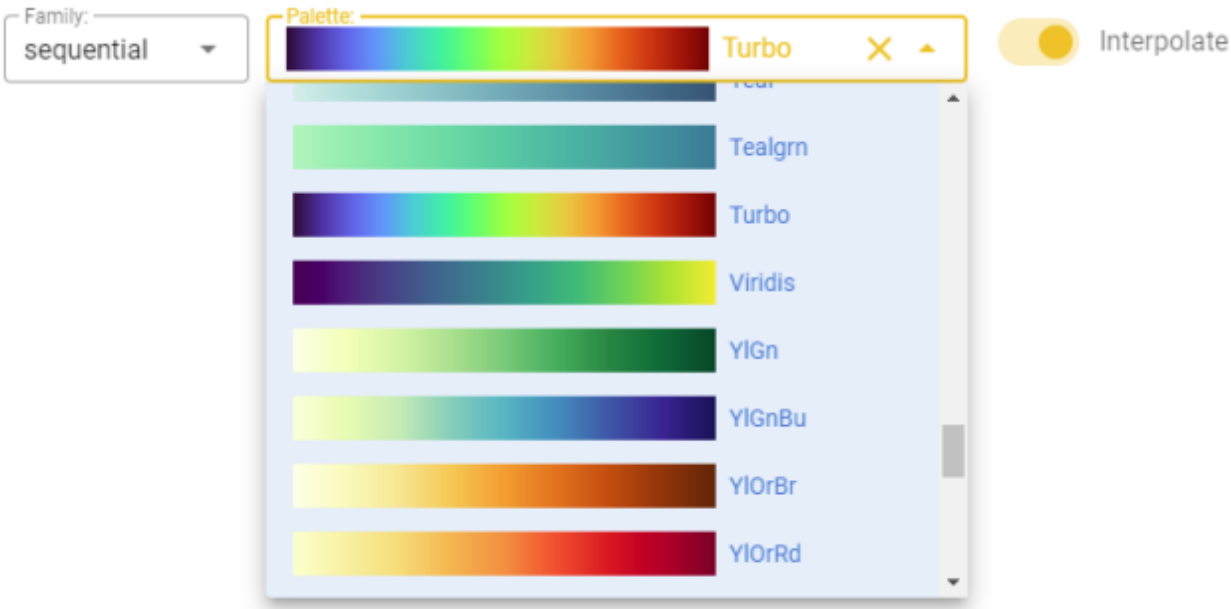


Fig. 3.56: Example of a palette picker that also manages the palette families

property colors

Get the colors of the selected palette.

Returns

colorlist – List of colors of the selected palette

Return type

list of strings in '#RRGGBB' format

Example

Get the selected palette colors:

```
print(picker.colors)
```

draw()

Returns the ipyvuetify object to display (the internal v.selectImage)

updatePalettes(*family='sequential'*, *interpolate=True*)

Update the displayed palette by changing family and/or interpolation flag

Parameters

- **family** (*str, optional*) – Family of the palette, one of these values: ['carto', 'cmocean', 'cyclical', 'diverging', 'plotlyjs', 'qualitative', 'sequential'] (default is 'sequential')
- **interpolate** (*bool, optional*) – If True the colors are displayed as interpolated (default is True)

property value

Get/Set name of the selected palette.

Returns
name – Name of the currently selected palette

Return type
str

Example

Set and then get the current palette name:

```
picker.value = 'Viridis'
print(picker.value)
```

3.3.2.25 palettePickerEx widget

Extended selection of a palette of different families (sequential, divergent, etc.)

class palettePickerEx.**palettePickerEx**(*family='sequential'*, *value='Viridis'*, *interpolate=True*, *show_interpolate_switch=True*, *onchange=None*, *width=400*, *clearable=True*, *show_opacity_slider=True*, *onchangeOpacity=None*)

Advanced selection of a palette of colors managing all the palette families and the interpolate flag

Parameters

- **family** (*str, optional*) – Family of the palette, one of these values: ['carto', 'cmocean', 'cyclical', 'diverging', 'plotlyjs', 'qualitative', 'sequential', 'custom'] (default is 'sequential')
- **value** (*str, optional*) – Name of the initially selected palette (default is 'Viridis')
- **interpolate** (*bool, optional*) – If True the colors are displayed as interpolated (default is True)

- **show_interpolate_switch** (*bool, optional*) – If True an interpolate switch is shown to enable or disable the interpolated display of the selected palette (default is True)
- **width** (*int, optional*) – Width of the widget in pixels (default is 400)
- **clearable** (*bool, optional*) – If True the dwopdown list of palettes will allow for no selection (default is True)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the palettes. The function will pass as parameters the list of colors and the interpolate flag (default is None)
- **show_opacity_slider** (*bool, optional*) – If True an slider to select opacity is shown (default is False)
- **onchangeOpacity** (*function, optional*) – Python function to call when the user changes the opacity. The function will as parameter the selected opacity in [0.0,1.0] (default is None)

Examples

Creation of a selection widget for the palettes managing all the families:

```
from vois.vuetify import palettePickerEx
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(colors, interpolate):
    with output:
        print(colors, interpolate)

p = palettePickerEx.palettePickerEx(onchange=onchange)

display(p.draw())
display(output)
```

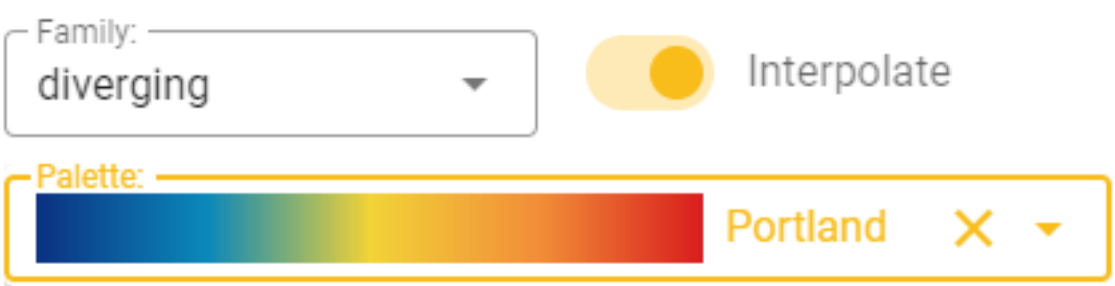


Fig. 3.57: Example of an extended palette picker managing all the palette families and the interpolate flag

property colors

Get the colors of the selected palette.

Returns

colorlist – List of colors of the selected palette

Return type
list of strings in '#RRGGBB' format

Example

Get the selected palette colors:

```
print(picker.colors)
```

property familyname

Get/Set name of the selected family.

Returns
name – Name of the currently selected family

Return type
str

Example

Set and then get the current palette family:

```
picker.familyname = 'qualitative'
print(picker.familyname)
```

property opacity

Get/Set the opacity value.

Returns
opacity – Current value of opacity in thenrange [0.0,1.0]

Return type
float

Example

Set and then get the opacity:

```
picker.opacity = 0.5
print(picker.opacity)
```

property value

Get/Set name of the selected palette.

Returns
name – Name of the currently selected palette

Return type
str

Example

Set and then get the current palette name:

```
picker.value = 'Viridis'
print(picker.value)
```

3.3.2.26 popup widget

Popup window opened at hover on a button.

class popup.**popup**(*widget*, *buttontext*, *buttonwidth=140*, *buttonheight=40*, *icon=None*, *icon_small=True*, *icon_large=False*, *margins=0*, *margintop=None*, *color='#f8bd1a'*, *rounded=True*, *outlined=True*, *text=False*, *popupwidth=160*, *popupheight=250*, *open_on_hover=True*, *close_on_click=True*, *close_on_content_click=True*, *title=''*, *show_close_button=False*)

Popup window opened at hover on a button

Parameters

- **widget** (*any widget*) – Widget to be displayed inside the popup menu
- **buttontext** (*str*) – Text to display inside the button
- **buttonwidth** (*int, optional*) – Width of the button in pixels (default is 140)
- **buttonheight** (*int, optional*) – Height of the button in pixels (default is 40)
- **icon** (*str, optional*) – Name of the icon to be displayed inside the button (default is None)
- **icon_small** (*bool, optional*) – If True the icon will be small (default is True)
- **icon_large** (*bool, optional*) – If True the icon will be small (default is False)
- **margins** (*int, optional*) – Dimension of the margins on all directions (default is 0)
- **margintop** (*int, optional*) – Dimension of the margin on top of the label (default is None)
- **color** (*str, optional*) – Color used for the button (default is the color_first defined in the settings.py module)
- **rounded** (*bool, optional*) – If True the button will be rounded (default is the button_rounded defined in the settings.py module)
- **outlined** (*bool, optional*) – If True the button will be outlined (default is True)
- **text** (*bool, optional*) – If True the button will display only the text and/or the icon, with no background (default is False)
- **popupwidth** (*int, optional*) – Width of the popup window in pixels (default is 160). The popup cannot have a width smaller than the width of the button
- **popupheight** (*int, optional*) – Height of the popup window in pixels (default is 250)
- **open_on_hover** (*bool, optional*) – If True the popup opens/closes when hovering the button, otherwise it opens/closes on click on the button (default is True)
- **close_on_click** (*bool, optional*) – Designates if popup should close on outside click (default is True)

- **close_on_content_click** (*bool, optional*) – Designates if popup should close when its content is clicked (default is True)
- **title** (*str, optional*) – Text to add at the top of the popup (default is '')
- **show_close_button** (*bool, optional*) – If True a close icon button is displayed on the top-right side of the popup to ease the closing of the popup (default is False). It can be useful mainly when close_on_click is set to False.

Note: All the icons from https://materialdesignicons.com/ site can be used, just by prepending 'mdi-' to their name.

All the free icons from https://fontawesome.com/ site can be used, just by prepending 'fa-' to their name.

Example

Creation and display of a popup window displaying a tree and opened by hovering on a button:

```
from vois.vuetify import popup, treeview
from IPython.display import display

sectors = ['S%d'%x for x in range(10)]
treecard = treeview.createTreeviewFromList(sectors,
                                           rootName='All',
                                           height='270px')

p = popup.popup(treecard, 'Sectors', popupheight=270)

display(p.draw())
```

draw()

Returns the ipyvuetify object to display

3.3.2.27 progress widget

Circular progress bar to use for lenghty operations.

class progress.**progress**(*output, text='', show=False, size=120, width=15, outputheight=400, color='#f8bd1a'*)

Circular progress bar to use for lenghty operations.

Parameters

- **output** (*ipywidgets.Output*) – Output widget on which the progress bar has to be displayed
- **text** (*str, optional*) – Small text to display inside the circle (default is '')
- **show** (*bool, optional*) – If True, display the progress upon creation (default is False)
- **size** (*int, optional*) – Diameter of the circle in pixels (default is 120 pixels)
- **width** (*int, optional*) – Width in pixels of the moving line (default is 15)

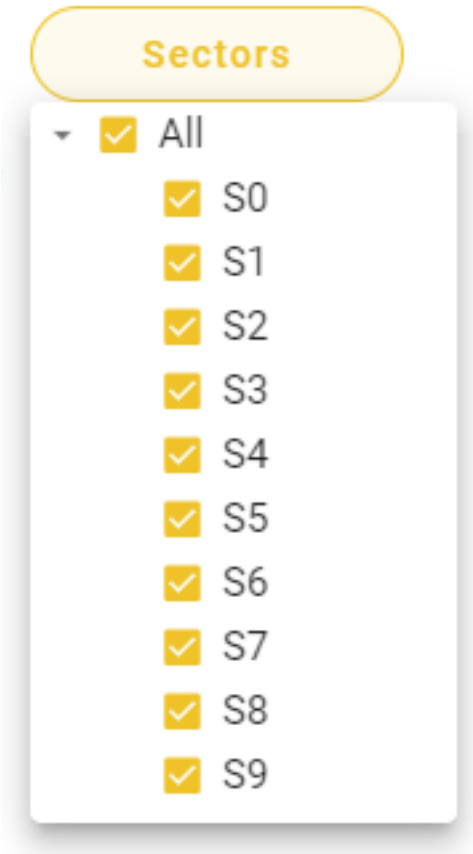


Fig. 3.58: Example of a button that, on hover, opens a popup window containing a treeview

- **outputheight** (*int, optional*) – Height in pixels of the output widget (default is 400 pixels)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the values in the list. The function will receive no parameter (use value property to retrieve the current selection)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)

Example

Creation and display of a progress widget:

```
from vois.vuetify import progress
from ipywidgets import widgets, Layout
from IPython.display import display

outputheight = 500
output = widgets.Output(layout=Layout(width='99%', height='%dpx' %␣
↪(outputheight+10)))
display(output)

p = progress.progress(output, text='Please, wait...',
                      size=200, width=20,
                      show=True, outputheight=outputheight)
```

To close the progress:

```
p.close()
```

Fig. 3.59: Example of progress widget displayed inside an Output.

close()

Hides the progress by clearing the output widget content

show()

Displays the progress into the output widget

showAbsolute(*output*, *left*, *top*, *zindex=9999*)

Displays the progress as an overlay layer in a specific position

Parameters

- **output** (*ipywidgets.Output*) – Output widget to use for the display of the transparent widget
- **left** (*str*) – Position of the left side of the widget. It can be in pixels, vw, or other units.
- **top** (*str*) – Position of the top side of the widget. It can be in pixels, vh, or other units.
- **zindex** (*int, optional*) – Z-index of the progress on the page (default is 9999)

3.3.2.28 queryStrings module

Read parameters passed in the URL of the Voila dashboard

queryStrings.**readParameters**()

Read parameters passed in the URL of the Voilà dashboard.

Returns

parameters – Dictionary containing key-values for all the URL passed to the Voilà page

Return type

dictionary

Example

Read URL parameters and get value of one of the parameters:

```
from vois.vuetify import queryStrings

parameters = queryStrings.readParameters()
activetab = parameters.get('activetab', ['chart'])[0]
print(activetab)
```

3.3.2.29 radio widget

Radio buttons to allow users to select from a predefined set of options.

class radio.**radio**(*index*, *labels*, *tooltips=None*, *color='#f8bd1a'*, *onchange=None*, *row=True*)

Radio buttons to allow users to select from a predefined set of options.

Parameters

- **index** (*int*) – Index of the option initially selected (from 0 to len(labels)-1)

- **labels** (*list of strings*) – Strings to be displayed as options in the radio widget
- **tooltips** (*list of str, optional*) – List of strings to use as tooltips for the corresponding radio items (default is None)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the values in the list. The function will receive a single parameter, containing the index of the selected option in the range from 0 to len(labels)-1
- **row** (*bool, optional*) – Flag to control the position of radio buttons, either horizontal or vertical (default is True)

Example

Creation and display of a radio widget to select among three options:

```
from vois.vuetify import radio
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(value):
    with output:
        print(value)

r = radio.radio(0,
                ['Option 0', 'Option 1', 'Option 2'],
                tooltips=['Tooltip for Option 1'],
                onchange=onchange,
                row=True)

display(r.draw())
display(output)
```

Fig. 3.60: Example of a radio widget to select from three options.

draw()

Returns the ipyvuetify object to display (the internal v.RadioGroup widget)

3.3.2.30 rangeSlider widget

Slider to select a range of numeric values.

class rangeSlider.**rangeSlider**(*selectedminvalue*, *selectedmaxvalue*, *minvalue*, *maxvalue*, *color='#f8bd1a'*, *onchange=None*, *height=250*, *vertical=True*)

Slider to select a range of numeric values.

Parameters

- **selectedminvalue** (*numeric*) – Minimum value of the selected range
- **selectedmaxvalue** (*numeric*) – Maximum value of the selected range
- **minvalue** (*numeric*) – Minimum value selectable by the user
- **maxvalue** (*numeric*) – Maximum value selectable by the user
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **onchange** (*function, optional*) – Python function to call when the user selects a value. The function will receive two parameters of numeric type containing the current min and max value selected by the user
- **height** (*int, optional*) – Height of the slider widget in pixel (default is 250 pixels)
- **vertical** (*bool, optional*) – Flag to display the range slider in vertical mode (default is True)

Example

Creation and display of a range slider widget:

```
from vois.vuetify import rangeSlider

s = rangeSlider.rangeSlider(5,18, 0,20)
s.draw()
```

Fig. 3.61: RangeSlider widget example

draw()

Returns the ipyvuetify object to display (the internal v.Html widget that contains a v.RangeSlider as its unique child)

3.3.2.31 rangeSliderFloat widget

Widget to select a range of float values

class rangeSliderFloat.**rangeSliderFloat**(*selectedminvalue*, *selectedmaxvalue*, *minvalue=0.0*, *maxvalue=1.0*, *text='Select'*, *showpercentage=True*, *decimals=2*, *maxint=None*, *labelwidth=0*, *sliderwidth=200*, *resetbutton=False*, *showtooltip=False*, *onchange=None*)

Compound widget to select a range of float values in a specific range.

Parameters

- **selectedminvalue** (*float*) – Initial minimum value of the slider
- **selectedmaxvalue** (*float*) – Initial maximum value of the slider
- **minvalue** (*float, optional*) – Minimum value of the slider (default is 0.0)
- **maxvalue** (*float, optional*) – Maximum value of the slider (default is 1.0)
- **text** (*str, optional*) – Text to display on the left of the slider (default is 'Select')
- **showpercentage** (*bool, optional*) – If True, the widget will write the current value as a percentage inside the allowed range and will append the % sign to the right of the current value (default is True)
- **decimals** (*int, optional*) – Number of decimal digits to use in case showpercentage is False (default is 2)
- **maxint** (*int, optional*) – Maximum integer number for the underlining integer slider (defines the slider sensitivity). Default value is None, meaning it will be automatically calculated
- **labelwidth** (*int, optional*) – Width of the label part of the widgets in pixels (default is 0, meaning it is automatically calculated based on the provided label text)
- **sliderwidth** (*int, optional*) – Width in pixels of the slider component of the widget (default is 200)
- **resetbutton** (*bool, optional*) – If True a reset button is displayed, allowing for resetting the widget to its initial value (default is False)
- **showtooltip** (*bool, optional*) – If True the up and down buttons will have a tooltip (default is False)
- **onchange** (*function, optional*) – Python function to call when the changes the range value of the slider. The function will receive a single parameter, containing the new value of the opacity in the range from 0.0 to 1.0 (default is None)

Example

Creation and display of a range slider widget to select an opacity range in a custom interval:

```
from vois.vuetify import rangeSliderFloat
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(value):
    with output:
        print(value)

o = rangeSliderFloat.rangeSliderFloat(1.8, 2.5, text='Select Value in [1.0,3.0]:',
                                      minvalue=1.0, maxvalue=3.0, onchange=onchange)

display(o.draw())
display(output)
```

Fig. 3.62: Example of an range slider widget to select a floating point range.

draw()

Returns the ipyvuetify object to display (a v.Row widget)

property value

Get/Set the slider range value as a list of min and max selected value.

Returns
value – Current value of the range slider

Return type
list of 2 floats

Example

Programmatically set the slider value

s.value = [0.56, 0.83]

3.3.2.32 selectImage widget

Select widget that displays images and enables for single selection.

class selectImage.**selectImage**(***kwargs: Any*)

Select widget that displays images and enables for single selection.

Parameters

- **images** (*list of json element, one for each image to display, optional*) – Each of the json elements must have "name", and "image" tags. Optional tags are "max_width", "max_height", "margins"
- **label** (*str, optional*) – Label to show in the select widget (default is '')
- **selection** (*int, optional*) – Index of the image to display as selected in the selection widgets at start (default is -1)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the images. The function will receive no parameters. (default is None)
- **color** (*str, optional*) – Main color of the selection widget (default is settings.color_first)
- **outlined** (*bool, optional*) – If True the selection widget will have a border around it (default is True)
- **clearable** (*Bool, optional*) – If True the selection widget will show a -x- button to clear the selection (default is True)
- **width** (*str, optional*) – Width of the select widget. It can be expressed as pixels (ex: "400px") or percentage (ex: "50%"). Default is "100%"
- **dense** (*Bool, optional*) – If True the items of the widgets have a reduced height (default is True)
- **max_width** (*int, optional*) – Max width in pixels of the images displayed inside the selection widget (default is 100)
- **max_height** (*int, optional*) – Max height in pixels of the images displayed inside the selection widget (default is 100)
- **margins** (*str, optional*) – Margins to apply to the images in the selection list and when displayed as selected in the widget (default is "ma-0 mr-1 mb-1")

Example

Creation of a select widget to select an image:

```
from vois.vuetify import selectImage
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange():
    with output:
        print(s.value)

images = [
```

(continues on next page)

(continued from previous page)

```
    { "name": 'Image 0', "image": 'https://cdn.vuetifyjs.com/images/cards/plane.
↪jpg'},
    { "name": 'Image 1', "image": 'https://cdn.vuetifyjs.com/images/cards/house.
↪jpg'},
    { "name": 'Image 2', "image": 'https://cdn.vuetifyjs.com/images/cards/road.jpg
↪'},
    { "name": 'Image 3', "image": 'https://cdn.vuetifyjs.com/images/cards/
↪sunshine.jpg'}
]

s = selectImage.selectImage(images=images, selection=1, max_height=100,␣
↪onchange=onchange,
                            label='Please select an image from the list',
                            outlined=True, clearable=True, margins="ma-1")

display(s)
display(output)
```

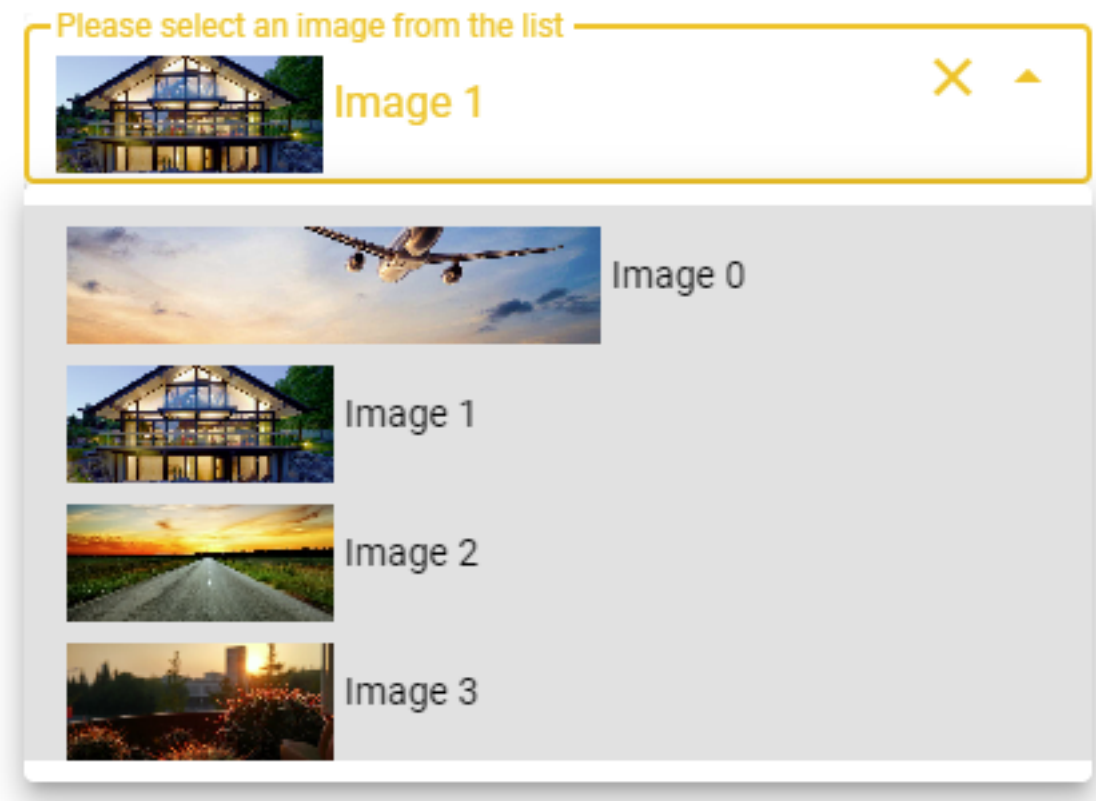


Fig. 3.63: Example of a selectImage widget

setImages(*images*)

Change the images to be selected.

Parameters

images (*list of json element, one for each image to display*) – Each of the json elements must have "name", and "image" tags. Optional tags are "max_width",

"max_height", "margins"

property value

Get/Set the selected image index.

Returns
v – index of the image currently selected

Return type
int

Example

Programmatically select one of the images given its index:

```
sel.value = 3
print(sel.value)
```

3.3.2.33 selectMultiple widget

Multiple selection widget.

class selectMultiple.**selectMultiple**(*label*, *values*, *selected=[]*, *mapping=None*, *reverse_mapping=None*, *width=300*, *onchange=None*, *marginy=1*)

Single selection widget from a dropdown list. Passing the parameter newvalues_enabled=True enables the user to insert new strings in the widget.

Parameters

- **label** (*str*) – Help text to display
- **values** (*list of strings*) – Strings to be displayed in the dropdown list of the widget
- **selected** (*list of str, optional*) – List of strings that are initially selected (default is [])
- **mapping** (*function, optional*) – Python function to call to transform the visible strings into codes (for example names of countries to their iso2 codes)
- **reverse_mapping** (*function, optional*) – Python function to call to transform the codes into visible strings (for example iso2 codes of countries to their names)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the values in the list. The function will receive no parameter (use value property to retrieve the current selection)
- **width** (*int, optional*) – Width in pixel of the widget (default is 300 pixels)
- **marginy** (*int, optional*) – Margin in y coordinates to position the widget from top (default is 1)

Example

Creation and display of a multiple selection widget for the selection of one or more countries:

```
from vois.vuetify import selectMultiple
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onchange():
    with output:
        print(sel.value)

sel = selectMultiple.selectMultiple('Country:',
                                    ['Belgium', 'France', 'Italy', 'Germany'],
                                    selected=['France', 'Italy'],
                                    width=200,
                                    onchange=onchange)
sel.draw()
```

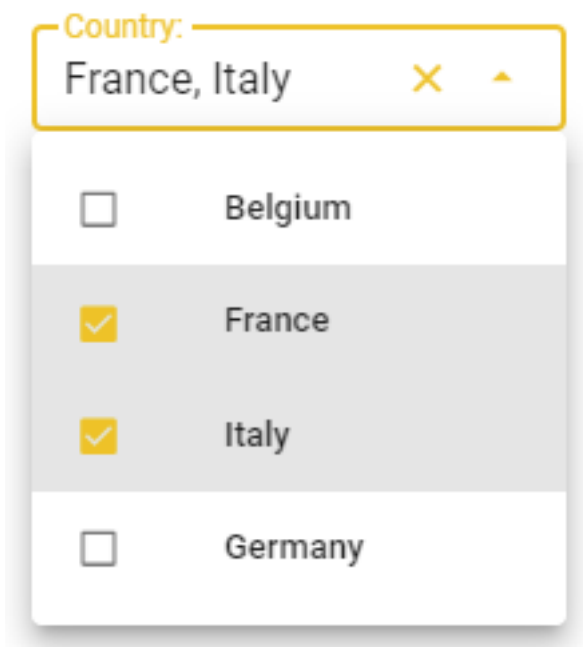


Fig. 3.64: Example of a selectMultiple widget to select from a list of countries.

draw()

Returns the ipyvuetify object to display (the internal v.Html widget that contains a v.Select as its unique child)

property value

Get/Set the selected value.

Returns

v – list of strings of the items currently selected

Return type
str

Example

Programmatically select a list of values:

```
sel.value = ['Italy', 'Belgium']
print(sel.value)
```

3.3.2.34 selectSingle widget

Single selection widget from a dropdown list.

class selectSingle.**selectSingle**(*label*, *values*, *selection=''*, *mapping=None*, *reverse_mapping=None*, *width=300*, *onchange=None*, *clearable=True*, *marginy=1*, *newvalues_enabled=False*, *newvalues_type='text'*, *colorbackground=False*)

Single selection widget from a dropdown list. Passing the parameter newvalues_enabled=True enables the user to insert new strings in the widget.

Parameters

- **label** (*str*) – Help text to display
- **values** (*list of strings*) – Strings to be displayed in the dropdown list of the widget
- **selection** (*str, optional*) – String that is initially selected (default is ‘’)
- **mapping** (*function, optional*) – Python function to call to transform the visible strings into codes (for example names of countries to their iso2 codes)
- **reverse_mapping** (*function, optional*) – Python function to call to transform the codes into visible strings (for example iso2 codes of countries to their names)
- **onchange** (*function, optional*) – Python function to call when the user selects one of the values in the list. The function will receive no parameter (use value property to retrieve the current selection)
- **width** (*int, optional*) – Width in pixel of the widget (default is 300 pixels)
- **clearable** (*bool, optional*) – Flag that controls if the widget should show to the user the ‘X’ button to clear its content (default is True)
- **marginy** (*int, optional*) – Margin in y coordinates to position the widget from top (default is 1)
- **newvalues_enabled** (*bool, optional*) – Flag that enables the user to insert new values beside those listed in the dropdown (default is False)
- **newvalues_type** (*str, optional*) – Type of the new values that the user can add in case newvalues_enabled is True (default is ‘text’)
- **colorbackground** (*bool, optional*) – Flag that controls the filling of the widget background with color when a value of the list is selected (default is False)

Example

Creation and display of a single select widget for the selection of a country:

```
from vois.vuetify import selectSingle
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onchange():
    with output:
        print(sel.value)

sel = selectSingle.selectSingle('Country:',
                                ['Belgium', 'France', 'Italy', 'Germany'],
                                selection='France',
                                width=200,
                                onchange=onchange)
sel.draw()
```

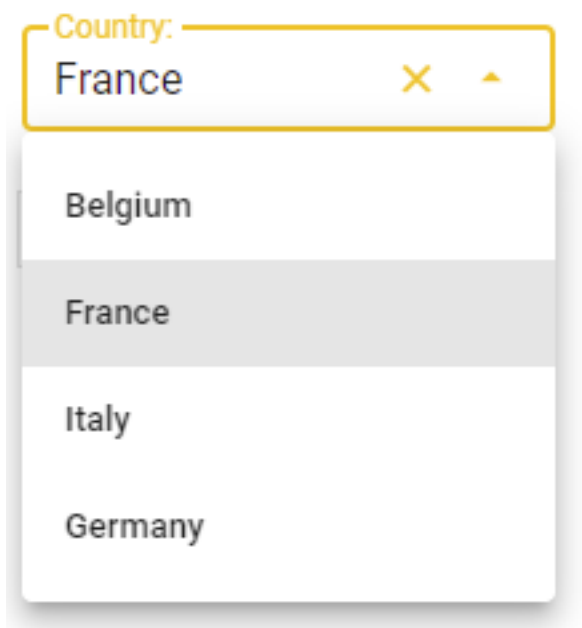


Fig. 3.65: Example of a selectSingle widget to select from a list of countries.

property disabled

Get/Set the disabled state.

draw()

Returns the ipyvuetify object to display (the internal v.Html widget that contains a v.Select or a v.Combobox as its unique child)

property value

Get/Set the selected value.

Returns
v – text of the item currently selected

Return type
str

Example

Programmatically select one value:

```
sel.value = 'Italy'
print(sel.value)
```

3.3.2.35 settings module

General settings for ipyvuetify widgets.

`settings.`**`button_rounded = True`**

Flag to control the appearance of all button widgets (also in the toggle module or inside the title and footer bars).

If True, the buttons will have rounded borders, if False, the buttons will have squared borders.

The *`button.button`* class has an optional parameter **rounded** that can be used to force a single button to be rounded or squared. This setting will have influence on all the buttons created without expliciting setting the **rounded** parameter.

Example of changing the appearance of all buttons to be rounded:

```
from vois.vuetify import settings, button
settings.button_rounded = True

import importlib
importlib.reload(button)

b = button.button('Round button', width=200, selected=True)
display(b.draw())
```

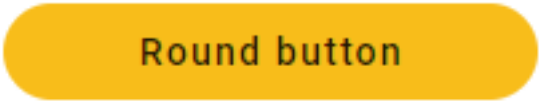


Fig. 3.66: Example of a round button

Example of changing the appearance of all buttons to be squared:

```
from vois.vuetify import settings, button
settings.button_rounded = False

import importlib
importlib.reload(button)
```

(continues on next page)

(continued from previous page)

```
b = button.button('Squared button', width=200, selected=True)
display(b.draw())
```

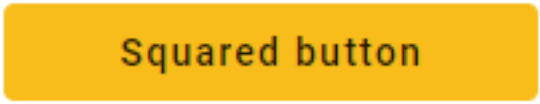


Fig. 3.67: Example of a squared button

Note: The usage of **importlib.reload** is needed to reload the button module after the settings.button_rounded flag is changed

`settings.color_first = '#f8bd1a'`

Primary color to be used for all the widgets of the vuetify package (for instance the buttons that have the selected state True will have this color as background color)

`settings.color_second = '#efefef'`

Secondary color to be used for all the widgets of the vuetify package (for instance the buttons that have the selected state False will have this color as background color)

`settings.dark_mode = False`

Flag to control the dark_mode of all the vuetify widgets.

If this flag is False, the foreground color used for text is 'black, otherwise it is 'white'

3.3.2.36 sidePanel widget

Side panel that opens on the side of the screen to show content or get user input.

class `sidePanel.`**`sidePanel`**(*title=''*, *text=''*, *width=500*, *right=False*, *zindex=9999*, *showclosebuttons=True*, *dark=False*, *backdark=False*, *content=[]*, *output=None*, *onclose=None*)

Side panel that opens on the side of the screen to show content or get user input.

Parameters

- **title** (*str, optional*) – Title to display on top of the side panel (default is '')
- **text** (*str, optional*) – String of text to display in the content of the side panel (default is '')
- **width** (*int, optional*) – Width of the panel in pixel (default is 500 pixels)
- **right** (*bool, optional*) – Flag that controls if the panel is opened on the right side of the screen (default is False)
- **zindex** (*int, optional*) – Z-index assigned to the panel (default is 9999)
- **showclosebuttons** (*bool, optional*) – Flag to display the close button in the top bar of the side panel (default is True)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)

- **backdark** (*bool, optional*) – Flag that controls the background color of the panel (if True, the background will be black, elsewhere white)
- **content** (*list of widgets, optional*) – Additional content to be added to the side panel to get user input (default is [])
- **output** (*ipywidgets.Output, optional*) – Output widget on which the side panel has to be displayed
- **onclose** (*function, optional*) – Python function to call when the user closes the side panel. The function will be called withput any parameter

Example

Creation and display of a side panel:

```
from vois.vuetify import sidePanel, slider
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onclose():
    with output:
        print('CLOSED')

text  = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod␣
↪tempor incididunt ut labore...'

s = slider.slider(66,0,100)

panel = sidePanel.sidePanel(title='Help panel', text=text,
                            width=400, content=[s.draw()],
                            output=output, onclose=onclose,
                            dark=False)
panel.show()
```

close(**args*)

Closes the side panel widget

isopen()

Queries the status of the side panel widget: returns True if the panel is displayed, False if it was closed

show()

Displays the side panel widget

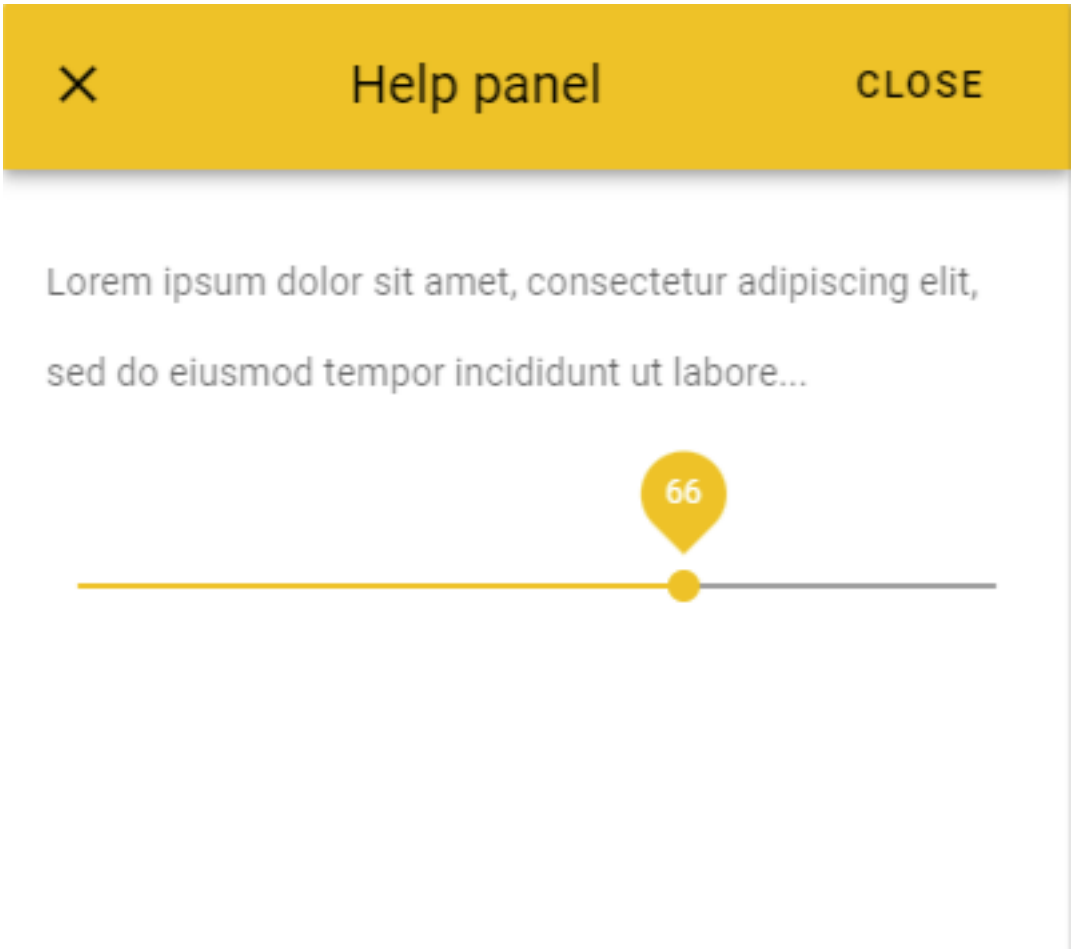


Fig. 3.68: Example of a side panel containing text and a slider widget.

3.3.2.37 slider widget

Slider widget is a better visualization of the number input. It is used for gathering numerical user data.

class slider.**slider**(*selectedvalue*, *minvalue*, *maxvalue*, *vertical=False*, *color='#f8bd1a'*, *onchange=None*, *height=120*, *width=None*, *step=1.0*)

Slider widget is a better visualization of the number input. It is used for gathering numerical user data.

Parameters

- **selectedvalue** (*numeric*) – Initial value of the slider
- **minvalue** (*numeric*) – Minimal value selectable by the user
- **maxvalue** (*numeric*) – Maximum value selectable by the user
- **step** (*numeric, optional*) – Step interval for ticks (default is 1.0)
- **vertical** (*bool, optional*) – Flag that controls the direction of the slider: horizontal or vertical (default is False)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **onchange** (*function, optional*) – Python function to call when the user selects a value. The function will receive a parameter of numeric type containing the current value of the slider widget
- **height** (*int, optional*) – Height of the slider widget in pixel (default is 120 pixels)
- **width** (*int, optional*) – Width of the slider widget in pixel. It is needed only for vertical sliders (default is None)

Example

Creation and display of a slider widget:

```
from vois.vuetify import slider
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onchange(value):
    with output:
        print(value)

s = slider.slider(2015, 2010,2021, onchange=onchange)
display(s.draw())
```

draw()

Returns the ipyvuetify object to display (the internal v.Html that contains a v.Slider widget as its only child)

property value

Get/Set the current value.

Returns

value – Current value selected

Fig. 3.69: Slider widget example

Return type
numeric

Example

Programmatically set the value and print it:

```
s.value = 2012
print(s.value)
```

3.3.2.38 sliderFloat widget

Widget to select a float value

class sliderFloat.**sliderFloat**(*value=1.0*, *minvalue=0.0*, *maxvalue=1.0*, *text='Select'*, *showpercentage=True*, *decimals=2*, *maxint=None*, *labelwidth=0*, *sliderwidth=200*, *resetbutton=False*, *showtooltip=False*, *onchange=None*)

Compound widget to select a float value in a specific range.

Parameters

- **value** (*float, optional*) – Initial value of the slider (default is 1.0)
- **minvalue** (*float, optional*) – Minimum value of the slider (default is 0.0)
- **maxvalue** (*float, optional*) – Maximum value of the slider (default is 1.0)
- **text** (*str, optional*) – Text to display on the left of the slider (default is 'Select')
- **showpercentage** (*bool, optional*) – If True, the widget will write the current value as a percentage inside the allowed range and will append the % sign to the right of the current value (default is True)
- **decimals** (*int, optional*) – Number of decimal digits to use in case showpercentage is False (default is 2)
- **maxint** (*int, optional*) – Maximum integer number for the underlining integer slider (defines the slider sensitivity). Default value is None, meaning it will be automatically calculated
- **labelwidth** (*int, optional*) – Width of the label part of the widgets in pixels (default is 0, meaning it is automatically calculated based on the provided label text)
- **sliderwidth** (*int, optional*) – Width in pixels of the slider component of the widget (default is 200)

- **resetbutton** (*bool, optional*) – If True a reset button is displayed, allowing for resetting the widget to its initial value (default is False)
- **showtooltip** (*bool, optional*) – If True the up and down buttons will have a tooltip (default is False)
- **onchange** (*function, optional*) – Python function to call when the changes the value of the slider. The function will receive a single parameter, containing the new value of the slider in the range from minvalue to maxvalue (default is None)

Example

Creation and display of a slider widget to select an opacity value in the range [0.0, 1.0]:

```
from vois.vuetify import sliderFloat
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(value):
    with output:
        print(value)

o = sliderFloat.sliderFloat(0.8, text='Fill opacity:', minvalue=0.0, maxvalue=1.0,␣
↪onchange=onchange)

display(o.draw())
display(output)
```

Fig. 3.70: Example of an slider widget to select a floating point value.

draw()

Returns the ipyvuetify object to display (a v.Row widget)

property value

Get/Set the slider value.

Returns
value – Current value of the slider

Return type
float

Example

Programmatically set the slider value

s.value = 0.56

3.3.2.39 snackbar widget

Widget to display a quick message to the user in an overlapping window that will disappear after a timeout

class snackbar.**snackbar**(*title=''*, *text=''*, *show=False*, *color='#f8bd1a'*, *dark=False*, *wrapwidth=80*, *timeout=10000*, *output=None*)

Widget to display a quick message to the user in an overlapping window that will disappear after a timeout.

Parameters

- **title** (*str, optional*) – Title of the message
- **text** (*str, optional*) – Text of the message
- **show** (*bool, optional*) – Flag to control the immediate show of the message to the user (default is True)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **wrapwidth** (*int optional*) – Number of chars for each line of the message (default is 80)
- **timeout** (*int, optional*) – Timeout in milliseconds before the widget disappears (default is 10000, 10 seconds)
- **output** (*ipywidgets.Output, optional*) – Output widget on which the snackbar has to be displayed

Example

Creation and display of a snackbar message:

```
from vois.vuetify import snackbar
from ipywidgets import widgets

title = 'Title of the message'
text  = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod␣
↪tempor incididunt ut labore...'

s = snackbar.snackbar(title=title, text=text, show=True, wrapwidth=40, timeout=3000)
s.draw()
```

close()

Closes the snackbar message

draw()

Returns the ipyvuetify object to display (the internal v.Snackbar widget)

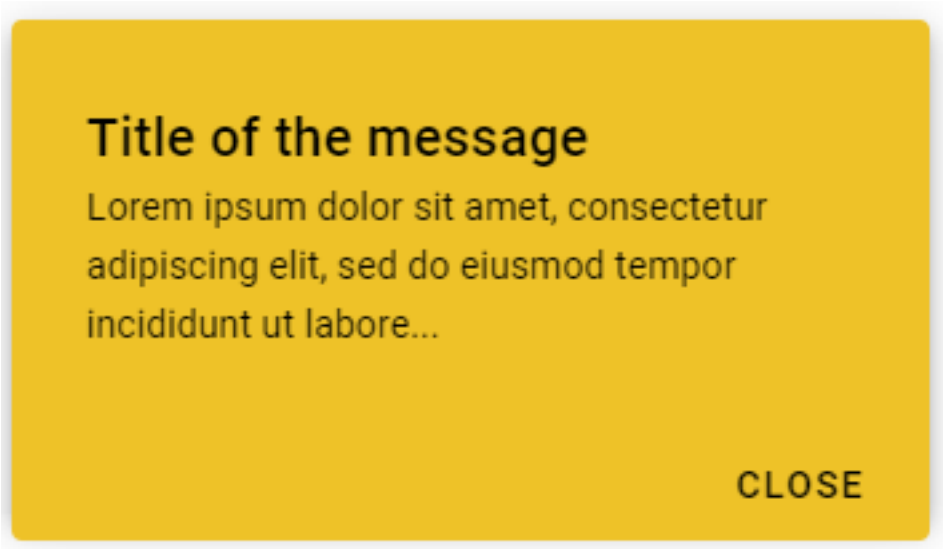


Fig. 3.71: Example of a snackbar message

show()

Shows the snackbar message

3.3.2.40 sortableList widget

Vertically aligned list of customizable cards with items that can be moved, added and removed.

class sortableList.**sortableList**(*items=[], width=400, maxheightlist=10000, outlined=True, dark=False, allowNew=True, newOnTop=False, newButtonOnTop=False, itemNew=None, itemContent=None, bottomContent=[], onchange=None, onmovedown=None, onmoveup=None, onremoving=None, onremoved=None, onadded=None, buttonstooltip=False, tooltipadd='Add new', tooltipdown='Move down', tooltipup='Move up', tooltipremove='Remove', activatable=False, ondeactivated=None, onactivated=None*)

Vertically aligned list of customizable cards with items that can be moved, added and removed.

Parameters

- **items** (*list of dicts, optional*) – List of items to display in the list (default is [])
- **width** (*int, optional*) – Width of widget in pixels (default is 400)
- **outlined** (*bool, optional*) – Flag to show each of the items with a border (default is True)
- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)
- **allowNew** (*bool, optional*) – If True, a 'plus' button is displayed that allows for adding new items (default is True)
- **newOnTop** (*bool, optional*) – If True, the '+' button adds a new item in first position on the items list (default False, new items are added as last in the items list)

- **newButtonOnTop** (*bool, optional*) – If True, the '+' button is displayed on top of the first item, otherwise it is displayed below the last item (default is False, the '+' button is on the bottom)
- **itemNew** (*function, optional*) – Python function called when a new items is added. The function is called with no arguments and it must return the dict initialized with the new item content (default is None). As an alternative, the function can return None, but then the real adding of the new item must be done by directly calling the doAddItem method
- **itemContent** (*function, optional*) – Python function called when an item is displayed. The function is called with an item as its first argument and the index (position of the item) as second argument. The function must return a list containing the ipyvuetify widgets to display the item content (default is None)
- **bottomContent** (*list of ipyvuetify widgets, optional*) – Additional widgets content to display in the bottom line (containing the 'plus' button), aligned to the right (default is [])
- **onchange** (*function, optional*) – Python function to call when the user changes the order of items or removes an item. The function will receive no parameters as input (default is None)
- **onmovedown** (*function, optional*) – Python function to call when the user moves one item down. The function will receive as parameter the zero-based index, before the move, of the moved item (default is None)
- **onmoveup** (*function, optional*) – Python function to call when the user moves one item up. The function will receive as parameter the zero-based index, before the move, of the moved item (default is None)
- **onremoving** (*function, optional*) – Python function to call when the user is about to remove an item. The function will receive as parameter the zero-based index of the item that is going to be removed (default is None)
- **onremoved** (*function, optional*) – Python function to call just after the user removes an item. The function will receive as parameter the zero-based index of the removed item (default is None)
- **onadded** (*function, optional*) – Python function to call just after a new item is added. The function will receive as parameter the zero-based index of the new item (default is None)
- **buttonstooltip** (*bool, optional*) – If True, the buttons to mode, add, remove items will have a tooltip (default is False)
- **tooltipadd** (*str, optional*) – Tooltip text for the "add" button (default is 'Add new')
- **tooltipdown** (*str, optional*) – Tooltip text for the "move down" buttons (default is 'Move down')
- **tooltipup** (*str, optional*) – Tooltip text for the "move up" buttons (default is 'Move up')
- **tooltipremove** (*str, optional*) – Tooltip text for the "remove" buttons (default is 'Remove')
- **activatable** (*bool, optional*) – If True the items can be activated by clicking on them (default is False)
- **ondeactivated** (*function, optional*) – Python function to call just after an item that was the active one, is deactivated. The function will receive as parameter the zero-based index of the deactivated item (default is None)

- **onactivated** (*function, optional*) – Python function to call just after an item becomes the active one (or by user-clicking or by setting the active property). The function will receive as parameter the zero-based index of the active item (default is None)

Examples

Simple list displaying static text:

```
from vois.vuetify import sortableList
from ipywidgets import widgets
from IPython.display import display
import ipyvuetify as v

items = [{ "name": 'Jane Adams',   "email": 'jane@adams.com'   },
         { "name": 'Paul Davis',   "email": 'paul@davis.com'   },
         { "name": 'Amanda Brown', "email": 'amanda@brown.com' }
        ]

# Creation of a new item
def itemNew():
    return {"name": "new", "email": "empty"}


# Content of an item
def itemContent(item, index):
    return [
        v.CardSubtitle(class_="mb-n4", children=[item['name']]),
        v.CardText(    class_="mt-n2", children=[item['email']])
    ]

s = sortableList.sortableList(items=items, dark=False, allowNew=True,
                              itemNew=itemNew, itemContent=itemContent)
display(s.draw())
```

Example of a sortable list displaying editable text, boolean and date values (using the *datePicker.datePicker* class):

```
from vois.vuetify import sortableList, datePicker, switch, tooltip
from ipywidgets import widgets
from IPython.display import display
import ipyvuetify as v

output = widgets.Output()

items = [
    { "name": "Paul",    "surname": "Dockery",  "married": False, "date": "" },
    { "name": "July",    "surname": "Winters",  "married": True,  "date": "1997-07-
↪28" },
    { "name": "David",   "surname": "Forest",   "married": True,  "date": "1999-03-
↪03" },
    { "name": "Dorothy", "surname": "Landmann", "married": False, "date": "" }
]
```

(continues on next page)

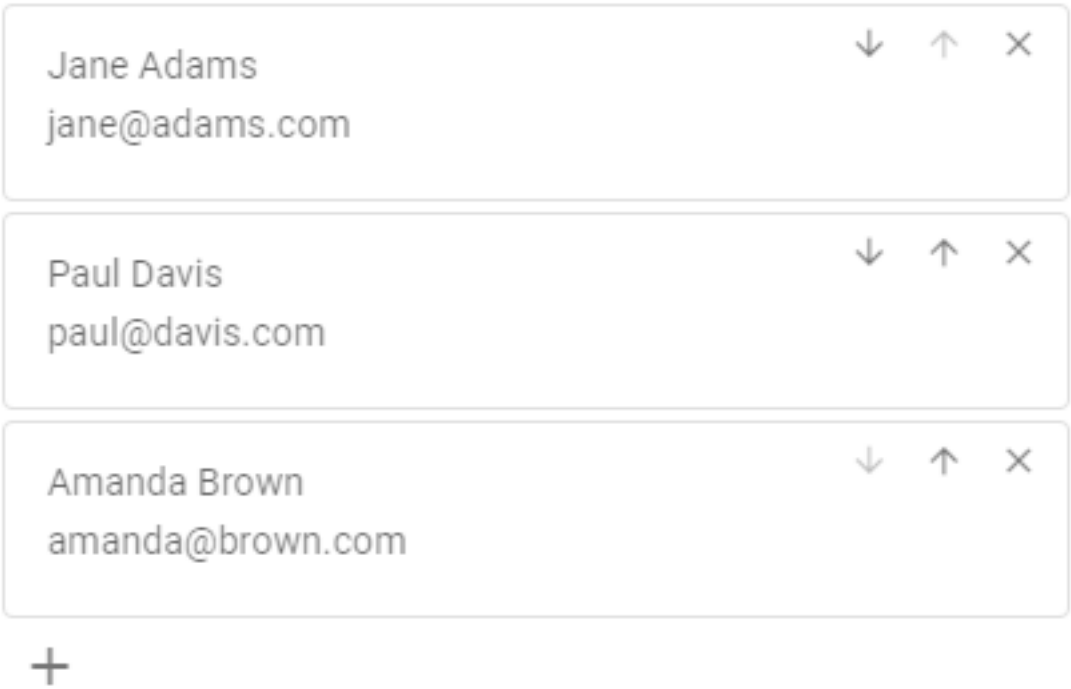


Fig. 3.72: Example of a simple sortableList displaying static text

(continued from previous page)

```
dark = False

# Called when an item is moved or deleted
def onchange():
    with output:
        print('Changed!')


# Creation of a new item
def itemNew():
    return { "name": "", "surname": "", "married": False, "date": "" }


# Remove all items
def itemRemoveAll(widget, event, data):
    s.items = []

reset = v.Btn(icon=True, children=[v.Icon(children=['mdi-playlist-remove'])])
reset.on_event('click', itemRemoveAll)


# Content of an item
def itemContent(item, index):

    def onname(widget, event, data):
        item["name"] = int(data)
```

(continues on next page)

(continued from previous page)

```
    def onsurname(widget, event, data):
        item["surname"] = data

    def onmarried(flag):
        item["married"] = flag
        dp.disabled = not flag
        if not flag:
            item["date"] = ''
            dp.date = None

    def ondate():
        item["date"] = dp.date

    tfname = v.TextField(label='Name:', value=item['name'],
                         color='amber', dense=True,
                         style_="max-width: 70px", class_="pa-0 ma-0 mt-2")
    tfname.on_event('input', onname)

    tfsurname = v.TextField(label='Surname:', value=item['surname'],
                            color='amber', dense=True,
                            style_="max-width: 100px", class_="pa-0 ma-0 mt-2")
    tfsurname.on_event('input', onsurname)

    sw = switch.switch(item['married'], "Married", onchange=onmarried)

    dp = datePicker.datePicker(date=item['date'], dark=dark, width=88,
                               onchange=ondate, offset_x=True, offset_y=False)
    dp.disabled = not item['married']

    sp = v.Html(tag='div', class_="pa-0 ma-0 mr-3", children=[' '])

    return [ v.Row(class_="pa-0 ma-0 ml-2", no_gutters=True,
                   children=[tfname, sp, tfsurname, sp, sw.draw(), sp, dp.draw()]) ]


s = sortableList.sortableList(items=items,
                              width=520,
                              outlined=False,
                              dark=dark,
                              allowNew=True,
                              itemNew=itemNew,
                              itemContent=itemContent,
                              bottomContent=[tooltip.tooltip("Remove all persons",
                                                             reset)],
                              onchange=onchange,
                              buttonstooltip=True)
display(s.draw())
display(output)
```

property active

Get/Set the active item.

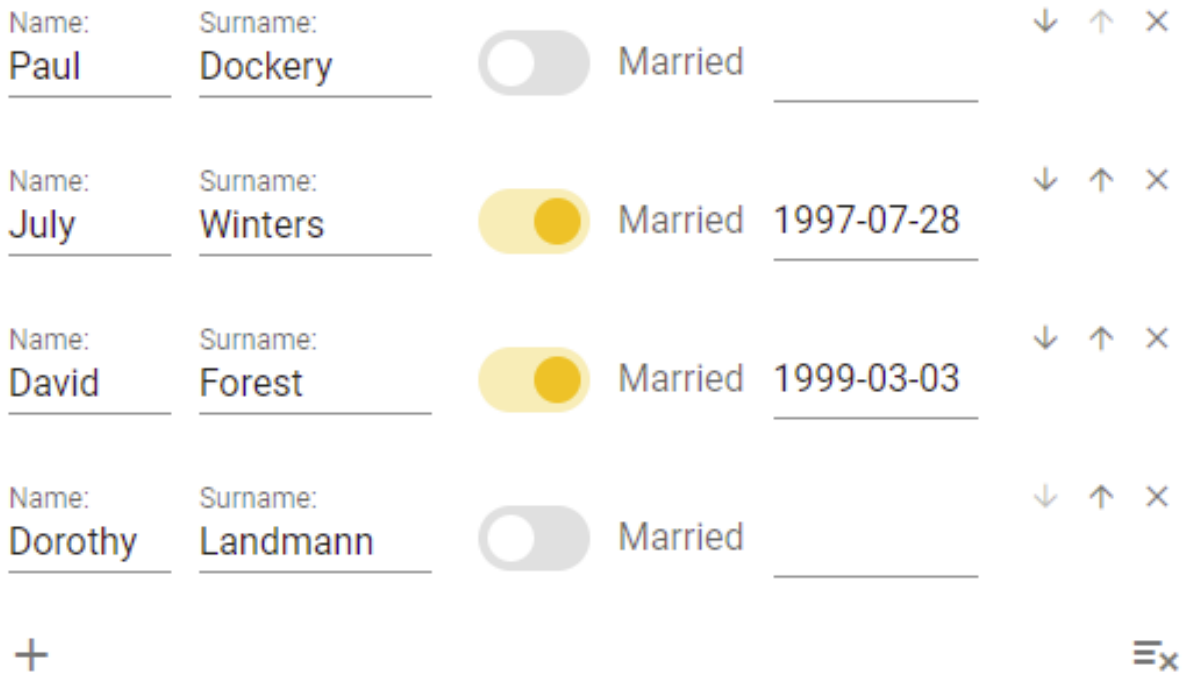


Fig. 3.73: Example of a sortableList to edit textual, boolean and date values on persons.

Returns
index – index of the active item

Return type
int

`doAddItem`(*item*)

Manual adding of a new item

`draw`()

Returns the ipyvuetify object to display (a v.Html object displaying two output widgets)

`property items`

Get/Set the updated items.

Returns
items – List of items in their updated position

Return type
list of dicts

3.3.2.41 svgsGrid widget

Display and selection of a list of SVG files

class svgsGrid.**svgsGrid**(***kwargs: Any*)

Display of a list of SVG files in rows and columns, with possibility to select one of the SVGs.

Parameters

- **filepaths** (*list of str*) – File paths of the SVG files to display in the grid
- **width** (*str, optional*) – Width to use for the display of the SVGs (default is '80px')
- **height** (*str, optional*) – Heigth to use for the display of the SVGs (default is '100px' to accomodate for the title of the files)
- **cols** (*int, optional*) – Horizontal column span [1,12] for each of the SVGs (default is 1, meaning 12 files are displayed in every row)
- **color** (*str, optional*) – Background color of the SVGs (default is 'white')
- **ripple** (*bool, optional*) – If True the click on the SVG files is highlighted (default is False)
- **svgsize** (*int, optional*) – Size in pixel of the square area where the SVG is displayed (default is 80)
- **on_click** (*function, optional*) – Python function to call when the user clicks on one of the SVG files. The function will receive as parameter the index of the clicked SVG. (default is None)

Example

Creation of a grid to display and select SVG files read from a subfolder:

```
from vois.vuetify import svgsGrid
from ipywidgets import widgets
from IPython.display import display
import glob

output = widgets.Output()

def on_click(index):
    with output:
        print(index)

filepaths = sorted(glob.glob("./maki/icons/*"))

svgsize = 80
height  = '%dpx'%(svgsize + 30)
g = svgsGrid(filepaths=filepaths, cols=1, svgsize=svgsize,
             height=height, ripple=True, on_click=on_click)

display(g)
display(output)
```

Fig. 3.74: Example of an svgsGrid to display a list of SVG files allowing selection

3.3.2.42 switch widget

The switch widget provides users the ability to choose between two distinct values.

class switch.**switch**(*flag*, *label*, *color='#f8bd1a'*, *inset=True*, *dense=False*, *onchange=None*)

The switch widget provides users the ability to choose between two distinct values.

Parameters

- **flag** (*int*) – Initial value of the switch
- **label** (*str*) – Text to display on the left of the switch widget
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **inset** (*bool, optional*) – Flag to enlarge switch track to encompass the thumb (default is True)
- **dense** (*bool, optional*) – If True, the widget is displayed with smaller dimensions (default is False)
- **onchange** (*function, optional*) – Python function to call when the user clicks on the switch. The function will receive a parameter of type bool containing the status of the switch flag

Example

Creation and display of a switch widget:

```
from vois.vuetify import switch
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(value):
    with output:
        print(value)

s = switch.switch(True, "Label of the switch", inset=True, onchange=onchange)

display(s.draw())
display(output)
```

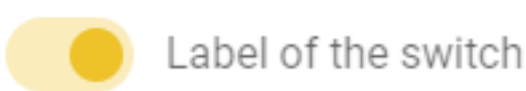


Fig. 3.75: Switch widget with inset flag True.

property disabled

Get/Set the disabled state.

draw()

Returns the ipyvuetify object to display (the internal v.Html that has a v.Switch widget as its only child)

property value

Get/Set the status of the switch.

Returns
flag – Status of the switch

Return type
bool

Example

Programmatically set the switch status and print it:

```
t.value = True
print(t.value)
```

3.3.2.43 tabs widget

Widget to select among alternative display using a list of tabs displayed horizontally or vertically.

class tabs.**tabs**(*index*, *labels*, *contents=None*, *tooltips=None*, *color='#f8bd1a'*, *dark=False*, *onchange=None*, *row=True*)

Widget to select among alternative display using a list of tabs displayed horizontally or vertically.

Parameters

- **index** (*int*) – Index of the selected option at start (from 0 to len(labels)-1)
- **labels** (*list of strings*) – Strings to be displayed as text of the options
- **contents** (*list of widgets, optional*) – Widgets to be alternatively displayed when each of the tabs option is selected (for instance could be a list of ipywidgets.Output widgets). Default is None
- **tooltips** (*list of strings, optional*) – List of strings to be used as tooltips for the single tabs, in order (default is None)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **dark** (*bool, optional*) – Flag to invert the text and backcolor (default is the value of settings.dark_mode)
- **onchange** (*function, optional*) – Python function to call when the user clicks on one of the tabs. The function will receive a parameter of type int containing the index of the selected tab, from 0 to len(labels)-1
- **row** (*bool, optional*) – Flag to display the tabs horizontally or vertically (default is True)

Example

Creation and display of a tabs widget to select among alternative Outputs display:

```
from vois.vuetify import tabs
from ipywidgets import widgets
from IPython.display import display

debug = widgets.Output()

output0 = widgets.Output()
output1 = widgets.Output()
output2 = widgets.Output()

with output0: print('This is output 0')
with output1: print('This is output 1')
with output2: print('This is output 2')

def onchange(index):
    with debug:
        print(index)

t = tabs.tabs(0, ['Option 0', 'Option 1', 'Option 2'],
              contents=[output0,output1,output2],
              onchange=onchange, row=False)

display(t.draw())
display(debug)
```

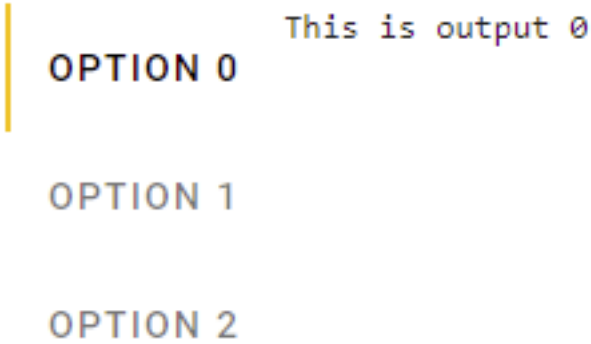


Fig. 3.76: Creation of a tabs widget to display ipywidgets.Output content at every selection.

property disabled

Get/Set the disabled state.

draw()

Returns the ipyvuetify object to display (the internal v.Tabs widget)

property value

Get/Set the active tab index.

Returns
index – Index of the selected tab (from 0 to len(labels)-1)

Return type
int

Example

Programmatically select one of the tab and print the index of the selected option:

```
t.value = 2
print(t.value)
```

3.3.2.44 textlist widget

Widget to display text strings vertically aligned.

class textlist.**textlist**(*titles*, *texts*, *titlesbold=[]*, *titlefontsize=12*, *textfontsize=12*, *titlecolumn=4*, *textcolumn=8*, *titlecolor='black'*, *textcolor='black'*, *lineheightfactor=1.5*)

Widget to vertically display a list of titles and texts strings. Each couple of title and text occupies a row.

Parameters

- **titles** (*list of strings*) – Strings to be displayed as title of each row
- **texts** (*list of strings*) – Strings to be displayed as the content of each row
- **titlesbold** (*list of strings, optional*) – List of titles whose corresponding texts should be displayed in with bold font (default is [])
- **titlefontsize** (*int, optional*) – Size in pixel of the font used for the titles (default is 12)
- **textfontsize** (*int, optional*) – Size in pixel of the font used for the texts (default is 12)
- **titlecolumn** (*int, optional*) – Number of column (out of 12) occupied by the titles (default is 4)
- **textcolumn** (*int, optional*) – Number of column (out of 12) occupied by the texts (default is 8)
- **titlecolor** (*str, optional*) – Color to use for the titles (default is 'black')
- **textcolor** (*str, optional*) – Color to use for the texts (default is 'black')
- **lineheightfactor** (*float, optional*) – Factor to multiply to the font-size to calculate the height of each row (default is 1.5)

Example

Creation and display of a widget to display some textual information:

```
from vois.vuetify import textlist
from IPython.display import display

t = textlist.textlist(['Name', 'Surname', 'Address', 'Role'],
                      ['Davide', 'De Marchi', 'via Eduardo 34, Roccacannuccia (PE)',
↪ 'Software developer'],
                      titlesbold=['Surname'],
                      titlefontsize=14,
                      textfontsize=16,
                      titlecolumn=3,
                      textcolumn=10,
                      titlecolor='#003300',
                      textcolor='#000000',
                      lineheightfactor=1.4
                     )

# Set some attributes of the card widget (margins, colors, width, etc.)
t.card.class_    = 'pa-0 ma-4 ml-6 mr-8'
t.card.flat      = False
t.card.color     = '#e0ffe0'
t.card.elevation = 8
t.card.width     = '430px'

display(t.draw())
```

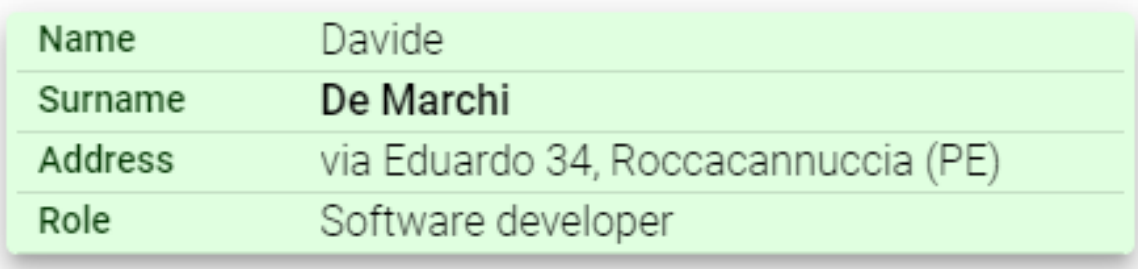


Fig. 3.77: Textlist widget for displaying textual information.

draw()

Returns the ipyvuetify object to display (the internal v.Card widget)

3.3.2.45 title bar

Class that implements a title bar that can be used as a simple main interface for a dashboard.

class title.**title**(*text=''*, *textweight=500*, *buttons=[]*, *menu=True*, *onmenuclick=None*, *dark=False*, *height=85*, *color='#f8bd1a'*, *onclick=None*, *logo='https://jeodpp.jrc.ec.europa.eu/services/shared/Notebooks/images/European_Commission.svg'*, *logowidth=100*, *logomarginy=0*, *menumarginy=2*, *output=None*)

Class that implements a title bar that can be used as a simple main interface for a dashboard.

Parameters

- **text** (*str, optional*) – Text to display in the title widget
- **textweight** (*int, optional*) – Weight of the title text (default is 500)
- **height** (*int, optional*) – Height of the title bar in pixels (default is 85 pixels)
- **color** (*str, optional*) – Background color of the title bar (default is the color_first defined in the settings.py module)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **buttons** (*list of strings, optional*) – List of strings to be used as text of the buttons to display below the title bar main text
- **onclick** (*function, optional*) – Python function to call when the user clicks on one of the buttons. The function will receive a parameter of string containing the name of the button clicked
- **menu** (*bool, optional*) – Flag that controls the display of a menu icon on the left side of the title bar (default is True)
- **menumarginy** (*int, optional*) – Vertical displace of the menu icon from the top of the title bar (default is 2)
- **onmenuclick** – Python function to call when the user clicks on the menu icon. The function will receive no parameters
- **logo** (*str, optional*) – String conaining the URL of the logo image to display on the right side of the title bar
- **logowidth** (*int, optional*) – Width in pixels of the area where the logo has to be displayed
- **logomarginy** (*int, optional*) – Vertical displace of the logo from the top of the title bar (default is 0)
- **output** (*ipywidgets.Output, optional*) – Output widget on which the title bar has to be displayed

Example

Creation and display of a title bar with additional buttons and left menu icon:

```
from vois.vuetify import title
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()
display(output)

def onclick(arg):
    with output:
        print(arg)

def onmenu():
    with output:
        print('MENU')

f = title.title(text='Title text to display', color='amber',
                menu=True, onmenuclick=onmenu,
                buttons=['Home', 'About Us', 'Team', 'Services', 'Blog', 'Contact Us
↪'],
                logo='https://jeodpp.jrc.ec.europa.eu/services/shared/home/images/
↪JRCBigDataPlatform2.png',
                logomarginy=6, menumarginy=4,
                height=80, onclick=onclick, output=output)
```

Fig. 3.78: Example of a title bar.

3.3.2.46 toggle widget

Widget to select among alternative options using a list of buttons displayed horizontally or vertically.

class toggle.**toggle**(*index*, *labels*, *tooltips=None*, *color='#f8bd1a'*, *onchange=None*, *row=True*, *width=150*, *height=36*, *justify='space-between'*, *rounded=True*, *outlined=False*, *colorselected='#f8bd1a'*, *colorunselected='#efefef'*, *dark=False*, *paddingrow=1*, *paddingcol=2*, *tile=False*, *small=False*, *xsmall=False*, *large=False*, *xlarge=False*)

Widget to select among alternative options using a list of buttons displayed horizontally or vertically.

Parameters

- **index** (*int*) – Index of the selected option at start (from 0 to len(labels)-1)
- **labels** (*list of strings*) – Strings to be displayed as text of the options
- **tooltips** (*list of strings, optional*) – Tooltip text for the options

- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **onchange** (*function, optional*) – Python function to call when the user clicks on one of the buttons. The function will receive a parameter of type int containing the index of the clicked button, from 0 to len(labels)-1
- **row** (*bool, optional*) – Flag to display the buttons horizontally or vertically (default is True)
- **width** (*int, optional*) – Width in pixels of the buttons (default is 150)
- **height** (*int, optional*) – Height in pixels of the buttons (default is 36)
- **justify** (*str, optional*) – In case of horizontal placement, applies the justify-content css property. Available options are: start, center, end, space-between and space-around.
- **rounded** (*bool, optional*) – Flag to display the buttons with a rounded shape (default is the button_rounded flag defined in the settings.py module)
- **outlined** (*bool, optional*) – Flag to display the buttons as outlined (default is False)
- **colorselected** (*str, optional*) – Color used for the buttons when they are selected (default is settings.color_first)
- **colorunselected** (*str, optional*) – Color used for the buttons when they are not selected (default is settings.color_second)
- **dark** (*bool, optional*) – Flag that controls the color of the text in foreground (if True, the text will be displayed in white, elsewhere in black)
- **paddingrow** (*int, optional*) – Horizontal padding among toggle buttons (1 unit means 4 pixels). Default is 1
- **paddingcol** (*int, optional*) – Vertical padding among toggle buttons (1 unit means 4 pixels). Default is 2
- **tile** (*bool, optional*) – Flag to remove the buttons small border (default is False)
- **large** (*bool, optional*) – Flag that sets the large version of the button (default is False)
- **xlarge** (*bool, optional*) – Flag that sets the xlarge version of the button (default is False)
- **small** (*bool, optional*) – Flag that sets the small version of the button (default is False)
- **xsmall** (*bool, optional*) – Flag that sets the xsmall version of the button (default is False)

Example

Creation and display of a widget for the selection among 3 options:

```
from vois.vuetify import toggle
from ipywidgets import widgets
from IPython.display import display

output = widgets.Output()

def onchange(index):
    with output:
        print(index)
```

(continues on next page)

(continued from previous page)

```
t = toggle.toggle(0, ['Option 1', 'Option 2', 'Option 3'], tooltips=['Tooltip for␣
↪option 1'],
                  onchange=onchange, row=False, width=150, rounded=False)

display(t.draw())
display(output)
```

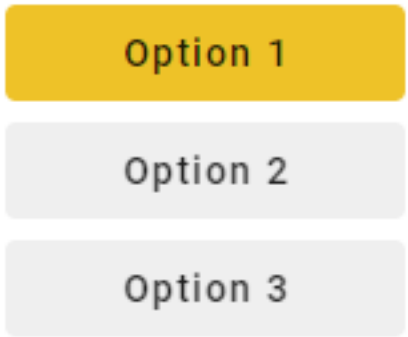


Fig. 3.79: Toogle widget for selecting alternative options using buttons.

draw()

Returns the ipyvuetify object to display (the internal v.Row or v.Col widget)

property value

Get/Set the active option index.

Returns

index – Index of the selected option (from 0 to len(labels)-1)

Return type

int

Example

Programmatically select one of the options and print the index of the selected option:

```
t.value = 2
print(t.value)
```

3.3.2.47 tooltip widget

Add tooltip text to a widget: returns a "modified" widget to be used instead of the original one.

`tooltip.`**tooltip**(*text*, *widget*)

Add a tooltip to a widget.

Parameters

- **text** (*str*) – Text of the tooltip

- **widget** (*ipyvuetify widget*) – Instance of the widget to which the tooltip has to be added

Returns
An ipyvuetify v.Item widget having a v.Tooltip as its only child

Return type
v.Item

Example

Add a tooltip to a switch widget:

```
from vois.vuetify import tooltip, switch
from IPython.display import display

s = switch.switch(True, "Activate the notification")
t = tooltip.tooltip('Select to activate the notification of events to the user', s.
↪draw())
display(t)
```

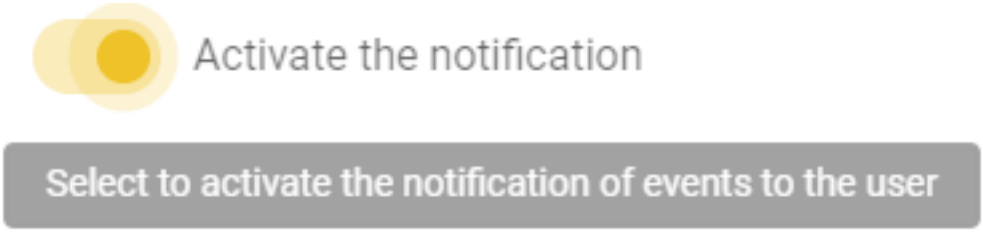


Fig. 3.80: Tooltip added to a switch widget.

3.3.2.48 treeview widget

Simplified creation of v-treeview vuetify widget to display hierarchical data in a tree

class treeview.**CustomTreeview**(***kwargs: Any*)

Ipyvuetify template to display a custom treeview. The nodes of the tree can have a checkbox (selectable=True) and/or can be activated (activatable=True).

items

JSON object to represent the tree: each object has 'id' and 'name' key, 'disabled', 'isfolder' and 'icon' are optional keys (default is [])

Type
list, optional

selectable

If True the nodes of the tree have a checkbox to select them (default is True)

Type
bool, optional

`activatable`

If True, one of the nodes of the tree can be activated (default is False)

Type
bool, optional

`selected`

List of id of the nodes that are selected on start (default is [])

Type
list, optional

`selectednames`

List of name of the nodes that are selected at any time (default is [])

Type
list, optional

`opened`

List of id of the nodes that are to be opened on start (default is [])

Type
list, optional

`color`

Color for the selected nodes (default is 'blue')

Type
str, optional

`on_change`

Python function to call when the selection of the tree items changes (default is None)

Type
function, optional

`on_activated`

Python function to call when the active item changes (user selecting a node) (default is None)

Type
function, optional

`expand_selection_to_parents`

If True, also the parent nodes are returned as selected when all children are selected (default is True)

Type
bool, optional

`iconsshow`

If True, an icon is added to each node of the tree (default is False)

Type
bool, optional

`iconscolor`

Color of the icons (default is 'blue')

Type
str, optional

icons_folder_opened

Name of the icon to use for opened nodes that have children when iconsfolder is True (default 'mdi-folder-open')

Type
str, optional

icons_folder_closed

Name of the icon to use for closed nodes that have children when iconsfolder is True (default 'mdi-folder')

Type
str, optional

tooltips

If True the nodes will show the tooltip (default is False)

Type
bool, optional

tooltips_chars

Minimum lenght of the node label to show the tooltip (default is 20)

Type
int, optional

search

Filter to display only the nodes of the tree that contain the text (default is '' which means that all the nodes of the tree are displayed)

Type
str, optional

item_height

Item height in pixels (default is 24)

Type
int, optional

font_size

Font size in pixels (default is 15)

Type
int, optional

icon_size

Icon size in pixels (default is 18)

Type
int, optional

checkbox_size

Checkbox size in pixels (default is 24)

Type
int, optional

Note: This class is not intended to be called directly, but only through the functions *createTreeviewFromList()* and *createTreeviewFromDF2Columns()*.

`treeview.`**`createFlatTreeview`**(*nameslist=[], select_all=True, on_change=None, on_activated=None, displayfullname=True, width='200px', height='500px', elevation=0, color='#f8bd1a', dark=False, transition=False, selectable=True, activatable=False, active=None, selected=[], disabled=[], iconsshow=False, iconscolor='#f8bd1a', iconsDict=None, tooltips=False, tooltips_chars=20, item_height=24, font_size=15, icon_size=18, checkbox_size=24*)

Create a flat treeview form a list of strings

Parameters

- **nameslist** (*list, optional*) – List of strings that contain a hierarchical structure, considering the separator character (default is [])
- **select_all** (*bool, optional*) – Flag to control the initial selection of all the nodes of the tree (default is True)
- **on_change** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking in one of the checkboxes (default is None)
- **on_activated** (*function, optional*) – Python function to call when the active item changes (user selecting a node) (default is None)
- **displayfullname** (*bool, optional*) – If True the nodes will display the full names, id False only the last part splitted by the separator (default is True)
- **width** (*str, optional*) – Width of the treeview widget (default is '200px')
- **height** (*str, optional*) – Height of the treeview widget (default is '500px')
- **elevation** (*int, optional*) – Elevation to assign to the widget (default is 0)
- **color** (*str, optional*) – Color to be used as main color of the Treeview widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the treeview widget will have a dark background (default is settings.dark_mode)
- **transition** (*bool, optional*) – If True applies a transition when nodes are opened and closed (default is False)
- **selectable** (*bool, optional*) – If True the nodes of the tree have a checkbox to select them (default is True)
- **activatable** (*bool, optional*) – If True, one of the nodes of the tree can be activated (default is False)
- **active** (*str, optional*) – Name of the node to activate on start (default is None)
- **selected** (*list of str, optional*) – List of fullnames of the nodes to select at start (default is []).
- **disabled** (*list of str, optional*) – List of fullnames of the nodes to display as disabled in the tree (default is [])
- **iconsshow** (*bool, optional*) – If True, an icon is added to each node of the tree (default is False)
- **iconscolor** (*str, optional*) – Color of the icons (default is settings.color_first)
- **iconsDict** (*dict, optional*) – Dictionary to apply icons to the fullnames of the items, if iconsshow is True (default is None)
- **tooltips** (*bool, optional*) – If True the nodes will show the tooltip (default is False)

- **tooltips_chars** (*int, optional*) – Minimum lenght of the node label to show the tooltip (default is 20)
- **item_height** (*int, optional*) – Item height in pixels (default is 24)
- **font_size** (*int, optional*) – Font size in pixels (default is 15)
- **icon_size** (*int, optional*) – Icon size in pixels (default is 18)
- **checkbox_size** (*int, optional*) – Checkbox size in pixels (default is 24)

Returns

v.Card – An ipyvuetify v.Card widget having a v.Html as its only child. The v.Html has a treeview.CustomTreeview widget as its only child

Return type

ipyvuetify Card widget

Example

Creation and display of a treeview:

```
from vois.vuetify import treeview
from IPython.display import display

treecard = treeview.createFlatTreeview(['A','B','C','D','E'], color='green', height=
↪'150px')
display(treecard)
```

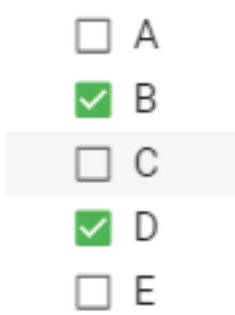


Fig. 3.81: Flat treeview widget created from a list of strings.

`treeview.`**`createTreeviewFromDF2Columns`**(*df, columnName1, columnName2, rootName='Root', separator='.', select_all=True, on_change=None, on_activated=None, expand_selection_to_parents=True, width='200px', height='500px', elevation=0, repeat_parent_as_first_child=False, color='#f8bd1a', dark=False, transition=False, selectable=True, activatable=False, active=None, open_on_click=False, opened=[], opened_all=False, selected=[], disabled=[], iconsshow=False, iconscolor='#f8bd1a', iconsfolder=True, icons_folder_opened='mdi-folder-open', icons_folder_closed='mdi-folder', iconroot=None, iconscolumnName1=None, iconscolumnName2=None, tooltips=False, tooltips_chars=20, item_height=24, font_size=15, icon_size=18, checkbox_size=24*)

Create a two levels treeview form the strings contained in two columns of a Pandas DataFrame.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame containing at least two columns of type string
- **columnName1** (*str*) – Name of the column of the Pandas DataFrame df containing the parent strings
- **columnName2** (*str*) – Name of the column of the Pandas DataFrame df containing the child strings
- **rootName** (*str, optional*) – Name to be displayed as root of the tree (default is 'Root')
- **separator** (*str, optional*) – String or character to be considered as separator for calculating the fullname of a node (default is '.'). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **select_all** (*bool, optional*) – Flag to control the initial selection of all the nodes of the tree (default is True)
- **on_change** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking in one of the checkboxes (default is None). The function receives as argument a list of all the fullnames of the selected nodes
- **on_activated** (*function, optional*) – Python function to call when the active item changes (user selecting a node) (default is None)
- **expand_selection_to_parents** (*bool, optional*) – If True, also the parent nodes are returned as selected when all children are selected (default is True)
- **width** (*str, optional*) – Width of the treeview widget (default is '200px')
- **height** (*int, optional*) – Height of the treeview widget (default is '500px')
- **elevation** (*int, optional*) – Elevation to assign to the widget (default is 0)
- **repeat_parent_as_first_child** (*bool, optional*) – If True each parent node will have a first children with its name (default is False)
- **color** (*str, optional*) – Color to be used as main color of the Treeview widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the treeview widget will have a dark background (default is settings.dark_mode)
- **transition** (*bool, optional*) – If True applies a transition when nodes are opened and closed (default is False)
- **selectable** (*bool, optional*) – If True the nodes of the tree have a checkbox to select them (default is True)
- **activatable** (*bool, optional*) – If True, one of the nodes of the tree can be activated (default is False)
- **active** (*str, optional*) – Fullname of the node to activate on start (default is None). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **open_on_click** (*bool, optional*) – If True, the nodes of the tree can be opened also by clicking on the node label (default is False)

- **opened** (*list of str, optional*) – List of fullnames of the nodes to open at start (default is []). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **opened_all** (*bool, optional*) – If True, all the tree nodes are opened at start (default is False). This setting has prevalence over the opened parameter.
- **selected** (*list of str, optional*) – List of fullnames of the nodes to select at start (default is []). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **disabled** (*list of str, optional*) – List of fullnames of the nodes to display as disabled in the tree (default is [])
- **iconsshow** (*bool, optional*) – If True, an icon is added to each node of the tree (default is False)
- **iconscolor** (*str, optional*) – Color of the icons (default is settings.color_first)
- **iconsfolder** (*bool, optional*) – If True (and if iconsshow is True) it adds the open/closed folder icon to nodes that have children (default is True)
- **icons_folder_opened** (*str, optional*) – Name of the icon to use for opened nodes that have children when iconsfolder is True (default 'mdi-folder-open')
- **icons_folder_closed** (*str, optional*) – Name of the icon to use for closed nodes that have children when iconsfolder is True (default 'mdi-folder')
- **iconroot** (*str, optional*) – Name of the icon for the root node of the tree (default is None)
- **iconscolumnName1** (*str, optional*) – Name of the DataFrame column that contains the icon name for the node on the first level of the tree (default is None)
- **iconscolumnName2** (*str, optional*) – Name of the DataFrame column that contains the icon name for the node on the second level of the tree (default is None)
- **tooltips** (*bool, optional*) – If True the nodes will show the tooltip (default is False)
- **tooltips_chars** (*int, optional*) – Minimum lenght of the node label to show the tooltip (default is 20)
- **item_height** (*int, optional*) – Item height in pixels (default is 24)
- **font_size** (*int, optional*) – Font size in pixels (default is 15)
- **icon_size** (*int, optional*) – Icon size in pixels (default is 18)
- **checkbox_size** (*int, optional*) – Checkbox size in pixels (default is 24)

Returns

An ipyvuetify v.Card widget having a v.Html as its only child. The v.Html has a treeview.CustomTreeview widget as its only child

Return type

v.Card

Example

Creation of a treeview from a simple Pandas DataFrame:

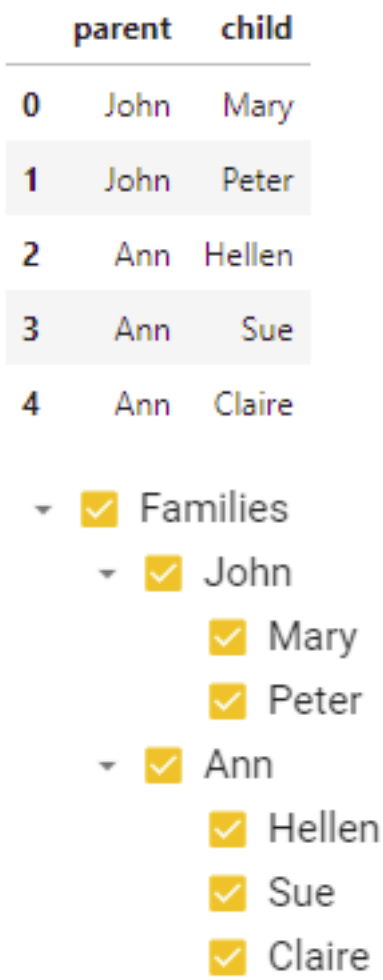
```
from vois.vuetify import treeview
import pandas as pd

table = [['parent', 'child'], ['John', 'Mary'], ['John', 'Peter'],
                              ['Ann', 'Hellen'], ['Ann', 'Sue'], ['Ann', 'Claire']]
headers = table.pop(0)
df = pd.DataFrame(table, columns=headers)
display(df)

def on_change(arg):
    print(arg)

treecard = treeview.createTreeviewFromDF2Columns(df, headers[0], headers[1],
                                                 rootName='Families',
                                                 on_change=on_change)
display(treecard)
```

treeview.**createTreeviewFromDF3Columns**(*df*, *columnName1*, *columnName2*, *columnName3*, *rootName='Root'*, *separator='.'*, *select_all=True*, *on_change=None*, *on_activated=None*, *expand_selection_to_parents=True*, *width='200px'*, *height='500px'*, *elevation=0*, *repeat_parent_as_first_child=False*, *color='#f8bd1a'*, *dark=False*, *transition=False*, *selectable=True*, *activatable=False*, *active=None*, *open_on_click=False*, *opened=[]*, *opened_all=False*, *selected=[]*, *disabled=[]*, *iconsshow=False*, *iconscolor='#f8bd1a'*, *iconsfolder=True*, *icons_folder_opened='mdi-folder-open'*, *icons_folder_closed='mdi-folder'*, *iconroot=None*, *iconscolumnName1=None*, *iconscolumnName2=None*, *iconscolumnName3=None*, *tooltips=False*, *tooltips_chars=20*, *item_height=24*, *font_size=15*, *icon_size=18*, *checkbox_size=24*)

Create a three levels treeview form the strings contained in three columns of a Pandas DataFrame.

Parameters

- **df** (*Pandas DataFrame*) – Pandas DataFrame containing at least two columns of type string
- **columnName1** (*str*) – Name of the column of the Pandas DataFrame df containing the parent strings
- **columnName2** (*str*) – Name of the column of the Pandas DataFrame df containing the child strings
- **columnName3** (*str*) – Name of the column of the Pandas DataFrame df containing the sub-child strings
- **rootName** (*str, optional*) – Name to be displayed as root of the tree (default is 'Root')
- **separator** (*str, optional*) – String or character to be considered as separator for calculating the fullname of a node (default is '.'). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **select_all** (*bool, optional*) – Flag to control the initial selection of all the nodes of the tree (default is True)



	parent	child
0	John	Mary
1	John	Peter
2	Ann	Hellen
3	Ann	Sue
4	Ann	Claire

Fig. 3.82: Treeview widget created from a Pandas DataFrame.

- **on_change** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking in one of the checkboxes (default is None). The function receives as argument a list of all the fullnames of the selected nodes
- **on_activated** (*function, optional*) – Python function to call when the active item changes (user selecting a node) (default is None)
- **expand_selection_to_parents** (*bool, optional*) – If True, also the parent nodes are returned as selected when all children are selected (default is True)
- **width** (*str, optional*) – Width of the treeview widget (default is '200px')
- **height** (*int, optional*) – Height of the treeview widget (default is '500px')
- **elevation** (*int, optional*) – Elevation to assign to the widget (default is 0)
- **repeat_parent_as_first_child** (*bool, optional*) – If True each parent node will have a first children with its name (default is False)
- **color** (*str, optional*) – Color to be used as main color of the Treeview widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the treeview widget will have a dark background (default is settings.dark_mode)
- **transition** (*bool, optional*) – If True applies a transition when nodes are opened and closed (default is False)
- **selectable** (*bool, optional*) – If True the nodes of the tree have a checkbox to select them (default is True)
- **activatable** (*bool, optional*) – If True, one of the nodes of the tree can be activated (default is False)
- **active** (*str, optional*) – Fullname of the node to activate on start (default is None). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **open_on_click** (*bool, optional*) – If True, the nodes of the tree can be opened also by clicking on the node label (default is False)
- **opened** (*list of str, optional*) – List of fullnames of the nodes to open at start (default is []). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **opened_all** (*bool, optional*) – If True, all the tree nodes are opened at start (default is False). This setting has prevalence over the opened parameter.
- **selected** (*list of str, optional*) – List of fullnames of the nodes to select at start (default is []). The full name of a node is the concatenation of the path to reach the node, using the separator character
- **disabled** (*list of str, optional*) – List of fullnames of the nodes to display as disabled in the tree (default is [])
- **iconsshow** (*bool, optional*) – If True, an icon is added to each node of the tree (default is False)
- **iconscolor** (*str, optional*) – Color of the icons (default is settings.color_first)
- **iconsfolder** (*bool, optional*) – If True (and if iconsshow is True) it adds the open/closed folder icon to nodes that have children (default is True)

- **icons_folder_opened** (*str, optional*) – Name of the icon to use for opened nodes that have children when iconsfolder is True (default 'mdi-folder-open')
- **icons_folder_closed** (*str, optional*) – Name of the icon to use for closed nodes that have children when iconsfolder is True (default 'mdi-folder')
- **iconroot** (*str, optional*) – Name of the icon for the root node of the tree (default is None)
- **iconscolumnName1** (*str, optional*) – Name of the DataFrame column that contains the icon name for the node on the first level of the tree (default is None)
- **iconscolumnName2** (*str, optional*) – Name of the DataFrame column that contains the icon name for the node on the second level of the tree (default is None)
- **iconscolumnName3** (*str, optional*) – Name of the DataFrame column that contains the icon name for the node on the third level of the tree (default is None)
- **tooltips** (*bool, optional*) – If True the nodes will show the tooltip (default is False)
- **tooltips_chars** (*int, optional*) – Minimum lenght of the node label to show the tooltip (default is 20)
- **item_height** (*int, optional*) – Item height in pixels (default is 24)
- **font_size** (*int, optional*) – Font size in pixels (default is 15)
- **icon_size** (*int, optional*) – Icon size in pixels (default is 18)
- **checkbox_size** (*int, optional*) – Checkbox size in pixels (default is 24)

Returns

An ipyvuetify v.Card widget having a v.Html as its only child. The v.Html has a treeview.CustomTreeview widget as its only child

Return type

v.Card

Example

Creation of a treeview from a simple Pandas DataFrame:

```
from vois.vuetify import treeview
import pandas as pd

table = [['parent', 'child', 'nephew'], ['John', 'Mary', 'Johnny'], ['John', 'Peter
↪', 'Lisa'],
                                        ['Ann', 'Hellen', 'July'], ['Ann', 'Sue',
↪'Hellen'],
                                        ['Ann', 'Claire', 'Pieter']]
headers = table.pop(0)
df = pd.DataFrame(table, columns=headers)
display(df)

def on_change(arg):
    print(arg)

treecard = treeview.createTreeviewFromDF3Columns(df, headers[0], headers[1],␣
↪headers[2],
```

(continues on next page)

(continued from previous page)

```
                                                    rootName='Families',
                                                    on_change=on_change)
display(treecard)
```

treeview.**createTreeviewFromList**(*nameslist=[], rootName='Root', separator='.', select_all=True, on_change=None, on_activated=None, expand_selection_to_parents=True, displayfullname=True, width='200px', height='500px', elevation=0, repeat_parent_as_first_child=False, substitutionDict=None, color='#f8bd1a', dark=False, transition=False, selectable=True, activatable=False, active=None, open_on_click=False, opened=[], opened_all=False, selected=[], disabled=[], iconsshow=False, iconscolor='#f8bd1a', iconsfolder=True, icons_folder_opened='mdi-folder-open', icons_folder_closed='mdi-folder', iconroot=None, iconsDict=None, tooltips=False, tooltips_chars=20, item_height=24, font_size=15, icon_size=18, checkbox_size=24*)

Create a treeview form a list of strings and a separator that defines the hierarchical structure (example: ['A', 'A.1', 'A.2', 'B', 'B.3']).

Parameters

- **nameslist** (*list, optional*) – List of strings that contain a hierarchical structure, considering the separator character (default is [])
- **rootName** (*str, optional*) – Name to be displayed as root of the tree (default is 'Root')
- **separator** (*str, optional*) – String or character to be considered as separator for extracting the hierarchical structure from the nameslist list of strings (default is '.')
- **select_all** (*bool, optional*) – Flag to control the initial selection of all the nodes of the tree (default is True)
- **on_change** (*function, optional*) – Python function to call when the selected nodes change caused by user clicking in one of the checkboxes (default is None)
- **on_activated** (*function, optional*) – Python function to call when the active item changes (user selecting a node) (default is None)
- **expand_selection_to_parents** (*bool, optional*) – If True, also the parent nodes are returned as selected when all children are selected (default is True)
- **displayfullname** (*bool, optional*) – If True the nodes will display the full names, id False only the last part splitted by the separator (default is True)
- **width** (*str, optional*) – Width of the treeview widget (default is '200px')
- **height** (*str, optional*) – Height of the treeview widget (default is '500px')
- **elevation** (*int, optional*) – Elevation to assign to the widget (default is 0)
- **repeat_parent_as_first_child** (*bool, optional*) – If True each parent node will have a first children with its name (default is False)
- **substitutionDict** (*dict, optional*) – Dictionary to apply substitutions to the fullnames of the items extracted from the nameslist parameters (default is None)
- **color** (*str, optional*) – Color to be used as main color of the Treeview widget (default is settings.color_first)
- **dark** (*bool, optional*) – If True, the treeview widget will have a dark background (default is settings.dark_mode)

	parent	child	nephew
0	John	Mary	Jhonny
1	John	Peter	Lisa
2	Ann	Hellen	July
3	Ann	Sue	Ellen
4	Ann	Claire	Pieter

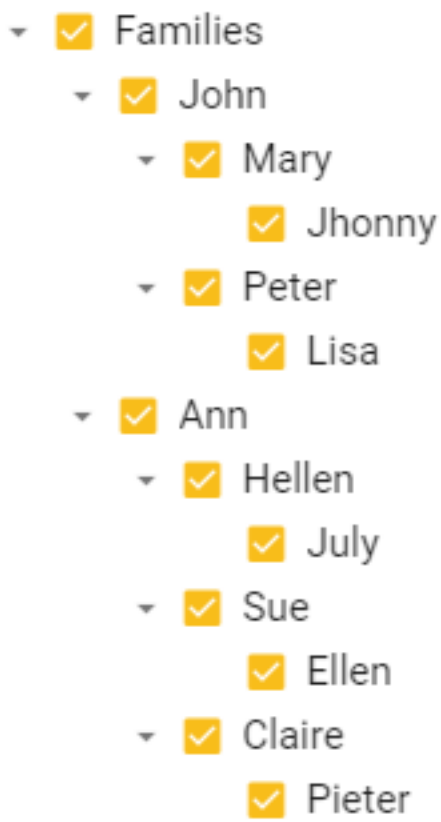


Fig. 3.83: Treeview widget created from a Pandas DataFrame.

- **transition** (*bool, optional*) – If True applies a transition when nodes are opened and closed (default is False)
- **selectable** (*bool, optional*) – If True the nodes of the tree have a checkbox to select them (default is True)
- **activatable** (*bool, optional*) – If True, one of the nodes of the tree can be activated (default is False)
- **active** (*str, optional*) – Name of the node to activate on start (default is None)
- **open_on_click** (*bool, optional*) – If True, the nodes of the tree can be opened also by clicking on the node label (default is False)
- **opened** (*list of str, optional*) – List of names of the nodes to open at start (default is [])
- **opened_all** (*bool, optional*) – If True, all the tree nodes are opened at start (default is False). This setting has prevalence over the opened parameter.
- **selected** (*list of str, optional*) – List of fullnames of the nodes to select at start (default is []).
- **disabled** (*list of str, optional*) – List of fullnames of the nodes to display as disabled in the tree (default is [])
- **iconsshow** (*bool, optional*) – If True, an icon is added to each node of the tree (default is False)
- **iconscolor** (*str, optional*) – Color of the icons (default is settings.color_first)
- **iconsfolder** (*bool, optional*) – If True (and if iconsshow is True) it adds the open/closed folder icon to nodes that have children (default is True)
- **icons_folder_opened** (*str, optional*) – Name of the icon to use for opened nodes that have children when iconsfolder is True (default 'mdi-folder-open')
- **icons_folder_closed** (*str, optional*) – Name of the icon to use for closed nodes that have children when iconsfolder is True (default 'mdi-folder')
- **iconroot** (*str, optional*) – Name of the icon for the root node of the tree (default is None)
- **iconsDict** (*dict, optional*) – Dictionary to apply icons to the fullnames of the items, if iconsshow is True (default is None)
- **tooltips** (*bool, optional*) – If True the nodes will show the tooltip (default is False)
- **tooltips_chars** (*int, optional*) – Minimum lenght of the node label to show the tooltip (default is 20)
- **item_height** (*int, optional*) – Item height in pixels (default is 24)
- **font_size** (*int, optional*) – Font size in pixels (default is 15)
- **icon_size** (*int, optional*) – Icon size in pixels (default is 18)
- **checkbox_size** (*int, optional*) – Checkbox size in pixels (default is 24)

Returns

v.Card – An ipyvuetify v.Card widget having a v.Html as its only child. The v.Html has a treeview.CustomTreeview widget as its only child

Return type

ipyvuetify Card widget

Example

Creation and display of a treeview:

```
from vois.vuetify import treeview
from IPython.display import display

treecard = treeview.createTreeviewFromList(['A','A.1','A.2','A.1.1',
                                            'A.3.1','A.4.1.2','A.5.2.1',
                                            'A.4.2.3.1','B','B.A'],
                                           rootName='Root',
                                           expand_selection_to_parents=False,
                                           substitutionDict={'A.1': 'A.1 new name'},
                                           color='green',
                                           height='350px')
display(treecard)
```

Fig. 3.84: Treeview widget created from a list of strings.

class treeview.**treeviewOperations**(*treecard*)

Helper class to operate on a treeview returned by the *createTreeviewFromList()* and *createTreeviewFromDF2Columns()* functions. This class allows for activation and opening/closing of nodes of the tree

treecard

Value returned by a call to the functions *createTreeviewFromList()* and *createTreeviewFromDF2Columns()*.

Type
instance of ipyvuetify Card widget

Example

Creation and display of a treeview and programmatical activation and opening of the nodes:

```
from vois.vuetify import treeview
from IPython.display import display

treecard = treeview.createTreeviewFromList(['A','A.1','A.2','A.1.1',
                                            'A.3.1','A.4.1.2','A.5.2.1',
                                            'A.4.2.3.1','B','B.A'],
                                           rootName='Root',
                                           activatable=True,
                                           expand_selection_to_parents=False,
                                           substitutionDict={'A.1': 'A.1 new name'},
                                           color='green',
                                           height=350)
display(treecard)

top = treeview.treeviewOperations(treecard)

# Print active node
print(top.getActive())

# Set active node
top.setActive('A.1.1')

# Set the list of opened nodes
top.setOpened(['Root', 'A', 'A.3'])
```

getActive()

Returns the fullname of the node that is active in the treeview

getChildren(*nodefullname*)

Returns the list of full names of the children of a node

getFirstChildFullname(*fullname*)

Given the fullname of a node, returns the fullname of its first child, or None if the node has no children

getOpened()

Returns the list of the fullnames of the opened nodes of the treeview

getSelected()

Returns the list of the fullnames of the selected nodes of the treeview

openAll()

Open all the nodes of the treeview

setActive(*fullname*)

Set the active node of the treeview by passing its fullname

setChildren(*nodefullname*, *childrenfullnames*)

Dynamically change the children of a node

Example

Creation and display of a treeview and programmatically add children to nodes:

```
from vois.vuetify import treeview
from IPython.display import display

treecard = treeview.createTreeviewFromList(['A','A.1','A.2','B'],
                                           separator='.',
                                           rootName='Root',
                                           expand_selection_to_parents=False,
                                           color='green',
                                           width='500px',
                                           height='350px',
                                           selectable=False,
                                           activatable=True)

display(treecard)

top = treeview.treeviewOperations(treecard)

top.setChildren('A.2', ['A.2.1', 'A.2.2'])
```

setOpened(*fullnames*)

Set the list of opened nodes of the treeview given their fullnames

setSearch(*text2search=''*)

Search for a text string in the nodes of the tree

setSelected(*fullnames*)

Set the list of selected nodes of the treeview given their fullnames

---

3.3.2.49 upload widget

Widget to upload files from the user local machine

class upload.**upload**(*accept=''*, *label=''*, *placeholder=''*, *color='#f8bd1a'*, *width='100%'*, *margins='pa-0 ma-0'*, *multiple=False*, *show_progress=True*, *onchange=None*)

Widget to upload files from the user local machine.

Parameters

- **accept** (*str, optional*) – String containing the comma separated list of mime types of files accepted for the upload operation (example: 'image/png, image/jpeg')
- **label** (*str, optional*) – Label displayed in the upper part of the widget (default is the empty string)

- **placeholder** (*str, optional*) – String that provides some guidance to the user (default is the empty string)
- **color** (*str, optional*) – Color used for the widget (default is the color_first defined in the settings.py module)
- **width** (*str, optional*) – Width of the control in pixels or in percentage (default is "100%")
- **margins** (*str, optional*) – Margins to apply to the widgets (default is "pa-0 ma-0")
- **multiple** (*bool, optional*) – Flag to enable multiple selection of files to upload (default is False)
- **show_progress** (*bool, optional*) – Flag to show a progress bar while uploading (default is True)
- **onchange** (*function, optional*) – Python function to call when the user selects one or more files to upload. The function will receive a parameter of type list containing the files to upload

Example

Creation and display of a widget for the upload of images:

```
from vois.vuetify import upload
from IPython.display import display

u = upload.upload(accept="image/png, image/jpeg, image/bmp",
                  label='Images',
                  placeholder='Click to select images to upload')
display(u.draw())
```

Fig. 3.85: Upload widget for selecting images to upload.

clear()

Sets the widget to its initial state (no file selected)

draw()

Returns the ipyvuetify object to display

Example

Display the upload widget:

```
import upload
u = upload.upload(accept="image/png, image/jpeg, image/bmp",
                  label='Images',
                  placeholder='Click to select images to upload')
display(u.draw())
```

vois library

CHAPTER

FOUR

SOURCES OF DOCUMENTATION AND HELP

This documentation is written using the sphinx tool.

4.1 Online documentation

This documentation is available in the source code under the docs directory (details on how to compile can be found in the README.md file). There is an online version in html format available on this website.

The documentation is divided in three main parts:

- *Introduction*: Introduction explaining the organization of the vois library package and its modules.
- *Tutorial*: A tutorial to guide you through the first steps of using vois library by using a step by step example dashboard.
- *Reference manual*: A manual describing all functions and methods available in vois library

4.2 Notebook examples

Each of the modules of the vois library is accompanied by a Jupyter Notebook having the same name in the examples/notebooks folder. In the notebook the functions of the module can be tested with simple lines of code that illustrate, through examples, how the classes and functions can be used.

Note: In the notebooks the python module to be tested can be imported in two different ways.

1. %run <modulename>.py

   This method "executes" the content of the python module in the notebook context. Classes and methods of the module can be called without prepending the modulename. This method is used to avoid the restart of the python kernel when the content of the python module is changed. By re-executing the notebook cell that contains the %run directive, the updated content is already available, without the need to shutdown and restart the notebook. It is suggested that this method is used only during the development of the vois library itself, and not in thedevelopment of users' dashboards.

   Example:

   ```
   %run button.py
   b = button('Test button')
   display(b.draw())
   ```

2. import <modulename>

This method imports the content of the python module: classes and methods of the module must be called by prepending the modulename.

Example:

```
import button
b = button.button('Test button')
display(b.draw())
```

4.3 Inline documentation

In addition to the online help pages, there is the inline documentation.

To get help on a class, e.g. *button.button*:

```
from vois.vuetify import button
help(button.button)
```

To get help on a class method, e.g., *app.app.snackbar()*:

```
from vois.vuetify import app
help(app.app.snackbar)
```

To get help on a function, e.g., *geojsonUtils.geojsonJson()*:

```
from vois import geojsonUtils
help(geojsonUtils.geojsonJson)
```

vois library

CHAPTER

FIVE

LICENSE

vois library

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

u

INDEX

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

U

V