Graph Objects in Python

Python classes that represent parts of a figure.

New to Plotly?

What Are Graph Objects?

The figures created, manipulated and rendered by the *plotly* Python library are represented by tree-like data structures which are automatically serialized to *JSON* for rendering by the Plotly.js JavaScript library. These trees are composed of named nodes called "attributes", with their structure defined by the Plotly.js figure schema, which is available in machine-readable form. **The plotly.graph_objects module (typically imported as go) contains an** automatically-generated hierarchy of Python classes **which represent non-leaf nodes in this figure schema. The term "graph objects" refers to instances of these classes.**

The primary classes defined in the **plotly.graph_objects** module are **Figure** and an **ipywidgets**-compatible variant called **FigureWidget**, which both represent entire figures. Instances of these classes have many convenience methods for Pythonically manipulating their attributes (e.g. **.update_layout()** or **.add_trace()**, which all accept "magic underscore" notation) as well as rendering them (e.g. **.show()**) and exporting them to various formats (e.g. **.to_json()** or **.write_image()** or **.write_html()**).

> Note: the functions in Plotly Express, which is the recommended entry-point into the **plotly** library, are all built on top of graph objects, and all return instances of **plotly.graph_objects.Figure**.

Every non-leaf attribute of a figure is represented by an instance of a class in the **plotly.graph_objects** hierarchy. For example, a figure **fig** can have an attribute **layout.margin**, which contains attributes **t**, **l**, **b** and **r** which are leaves of the tree: they have no children. The field at **fig.layout** is an object of class **plotly.graph_objects.Layout** and **fig.layout.margin** is an object of class **plotly.graph_objects.layout.Margin** which represents the **margin** node, and it has fields **t**, **l**, **b** and **r**, containing the values of the respective leaf-nodes. Note that specifying all of these values can be done without creating intermediate objects using "magic underscore" notation: **go.Figure(layout_margin=dict(t=10, b=10, r=10, l=10))**.

The objects contained in the list which is the value of the attribute **data** are called "traces", and can be of one of more than 40 possible types, each of which has a corresponding class in **plotly.graph_objects**. For example, traces of type **scatter** are represented by instances of the class **plotly.graph_objects.Scatter**. This means that a figure constructed as **go.Figure(data=[go.Scatter(x=[1,2], y=[3,4])])** will have the *JSON* representation **{"data": [{"type": "scatter", "x": [1,2], "y": [3,4]}]}**.

Graph Objects Compared to Dictionaries

Graph objects have several benefits compared to plain Python dictionaries:

1. Graph objects provide precise data validation. If you provide an invalid property name or an invalid property value as the key to a graph object, an exception will be raised with a helpful error message describing the problem. This is not the case if you use plain Python dictionaries and lists to build your figures.
2. Graph objects contain descriptions of each valid property as Python docstrings, with a full API reference available. You can use these docstrings in the development environment of your choice to learn about the available properties as an alternative to consulting the online Full Reference.
3. Properties of graph objects can be accessed using both dictionary-style key lookup (e.g. **fig["layout"]**) or class-style property access (e.g. **fig.layout**).
4. Graph objects support higher-level convenience functions for making updates to already constructed figures (**.update_layout()**, **.add_trace()** etc).
5. Graph object constructors and update methods accept "magic underscores" (e.g. **go.Figure(layout_title_text="The Title")** rather than **dict(layout=dict(title=dict(text="The Title")))**) for more compact code.
6. Graph objects support attached rendering (**.show()**) and exporting functions (**.write_image()**) that automatically invoke the appropriate functions from the plotly.io module.

When to use Graph Objects vs Plotly Express

The recommended way to create figures is using the functions in the plotly.express module, collectively known as Plotly Express, which all return instances of **plotly.graph_objects.Figure**, so every figure produced with the **plotly** library actually uses graph objects under the hood, unless manually constructed out of dictionaries.

That said, certain kinds of figures are not yet possible to create with Plotly Express, such as figures that use certain 3D trace-types like **mesh** or **isosurface**. In addition, certain figures are cumbersome to create by starting from a figure created with Plotly Express, for example figures with subplots of different types, dual-axis plots, or faceted plots with multiple different types of traces. To construct such figures, it can be easier to start from an empty **plotly.graph_objects.Figure** object (or one configured with subplots via the make_subplots() function) and progressively add traces and update attributes as above. Every **plotly** documentation page lists the Plotly Express option at the top if a Plotly Express function exists to make the kind of chart in question, and then the graph objects version below.

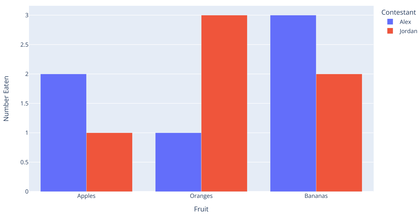
Note that the figures produced by Plotly Express **in a single function-call** are easy to customize at creation-time, and to manipulate after creation using the **update_*** and **add_*** methods.

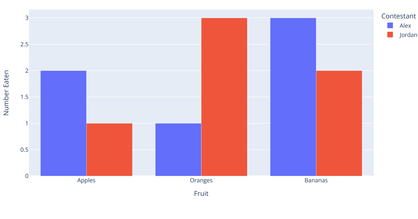
Comparing Graph Objects and Plotly Express

The figures produced by Plotly Express can always be built from the ground up using graph objects, but this approach typically takes **5-100 lines of code rather than 1**.

Here is a simple example of how to produce the same figure object from the same data, once with Plotly Express and once without. The data in this example is in "long form" but Plotly Express also accepts data in "wide form" and the line-count savings from Plotly Express over graph objects are comparable. More complex figures such as sunbursts, parallel coordinates, facet plots or animations require many more lines of figure-specific graph objects code, whereas switching from one representation to another with Plotly Express usually involves changing just a few characters.

```
import pandas as pd

df = pd.DataFrame({
  "Fruit": ["Apples", "Oranges", "Bananas", "Apples", "Oranges", "Bananas"],
  "Contestant": ["Alex", "Alex", "Alex", "Jordan", "Jordan", "Jordan"],
  "Number Eaten": [2, 1, 3, 1, 3, 2],
})


# Plotly Express

import plotly.express as px

fig = px.bar(df, x="Fruit", y="Number Eaten", color="Contestant", barmode="group")
fig.show()


# Graph Objects

import plotly.graph_objects as go

fig = go.Figure()
for contestant, group in df.groupby("Contestant"):
    fig.add_trace(go.Bar(x=group["Fruit"], y=group["Number Eaten"], name=contestant,
        hovertemplate="Contestant=%{x}<br>Fruit=%{y}<br>Number Eaten=%{y}<extra></extra>" % contestant))
fig.update_layout(legend_title_text = "Contestant")
fig.update_xaxes(title_text="Fruit")
fig.update_yaxes(title_text="Number Eaten")
fig.show()
```





What About Dash?

Dash is an open-source framework for building analytical applications, with no Javascript required, and it is tightly integrated with the Plotly graphing library.

Learn about how to install Dash at https://dash.plot.ly/installation.

Everywhere in this page that you see **fig.show()**, you can display the same figure in a Dash application by passing it to the **figure** argument of the **Graph** component from the built-in **dash_core_components** package like this:

```
import plotly.graph_objects as go # or plotly.express as px
fig = go.Figure() # or any Plotly Express function e.g. px.bar(...)
# fig.add_trace( ... )
# fig.update_layout( ... )

import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()
app.layout = html.Div([
    dcc.Graph(figure=fig)
])

app.run_server(debug=True, use_reloader=False)  # Turn off reloader if inside Jupyter
```