✨ Introducing Plotly Express ✨

plotly
Mar 20 · 11 min read

Plotly Express is a new high-level Python visualization library: it's a wrapper for Plotly.py that exposes a simple syntax for complex charts. Inspired by Seaborn and ggplot2, it was specifically designed to have a terse, consistent and easy-to-learn API: with just a single import, you can make richly interactive plots in just a single function call, including faceting, maps, animations, and trendlines. It comes with on-board datasets, color scales and themes, and just like Plotly.py, Plotly Express is *totally free*: with its permissive open-source MIT license, you can use it however you like (yes, even in commercial products!). Best of all, Plotly Express is fully compatible with the rest of Plotly ecosystem: use it in your Dash apps, export your figures to almost any file format using Orca, or edit them in a GUI with the JupyterLab Chart Editor!

If you're the **TL;DR** type, just `pip install plotly_express` and head on over to our walkthrough notebook or gallery or reference documentation to start playing around, otherwise read on for an overview of what makes Plotly Express special. If you have any feedback or want to check out the code, it's all up on Github.

Quick and easy data visualization with Plotly Express

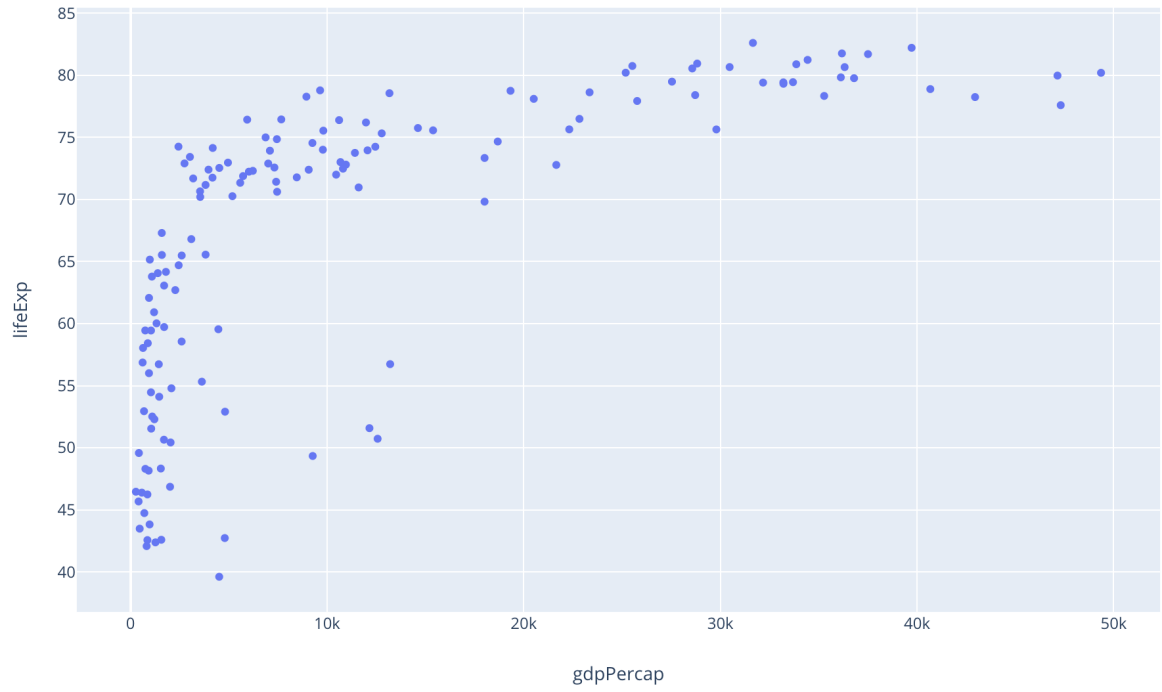
The code used to generate the screenshots below is available in our walkthrough notebook which you can load up on Binder to play with right now in your browser without installing anything.

Once you import Plotly Express (usually as `px`), most plots are made with just one function call that accepts a tidy Pandas data frame, and a simple description of the plot you want to make. If you want a basic scatter plot, it's just `px.scatter(data, x="column_name", y="column_name")`.

Here's an example with the Gapminder dataset – which comes built-in! – showing life expectancy vs GPD per capita by country for 2007:

```
import plotly_express as px
gapminder = px.data.gapminder()
gapminder2007 = gapminder.query("year == 2007")

px.scatter(gapminder2007, x="gdpPercap", y="lifeExp")
```

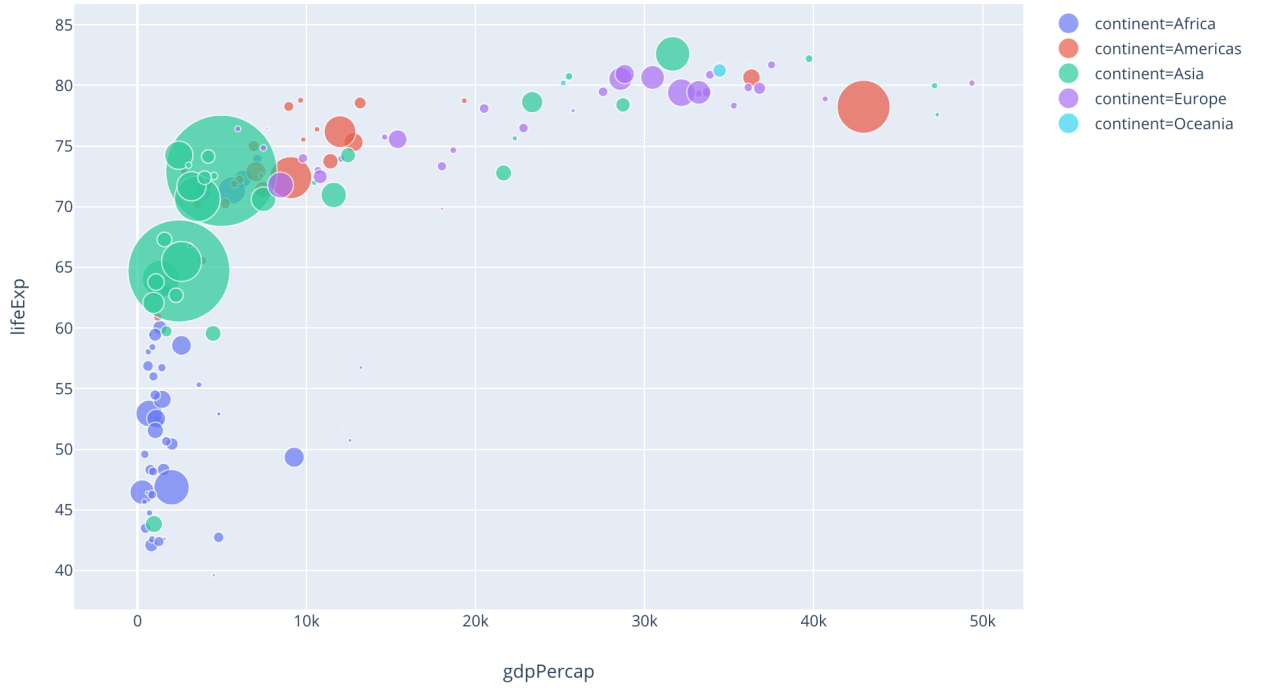


If you want to break that down by continent, you can color your points with the `color` argument and `px` takes care of the details, assigning default colors, setting up the legend etc:

```
px.scatter(gapminder2007, x="gdpPercap", y="lifeExp", color="continent")
```

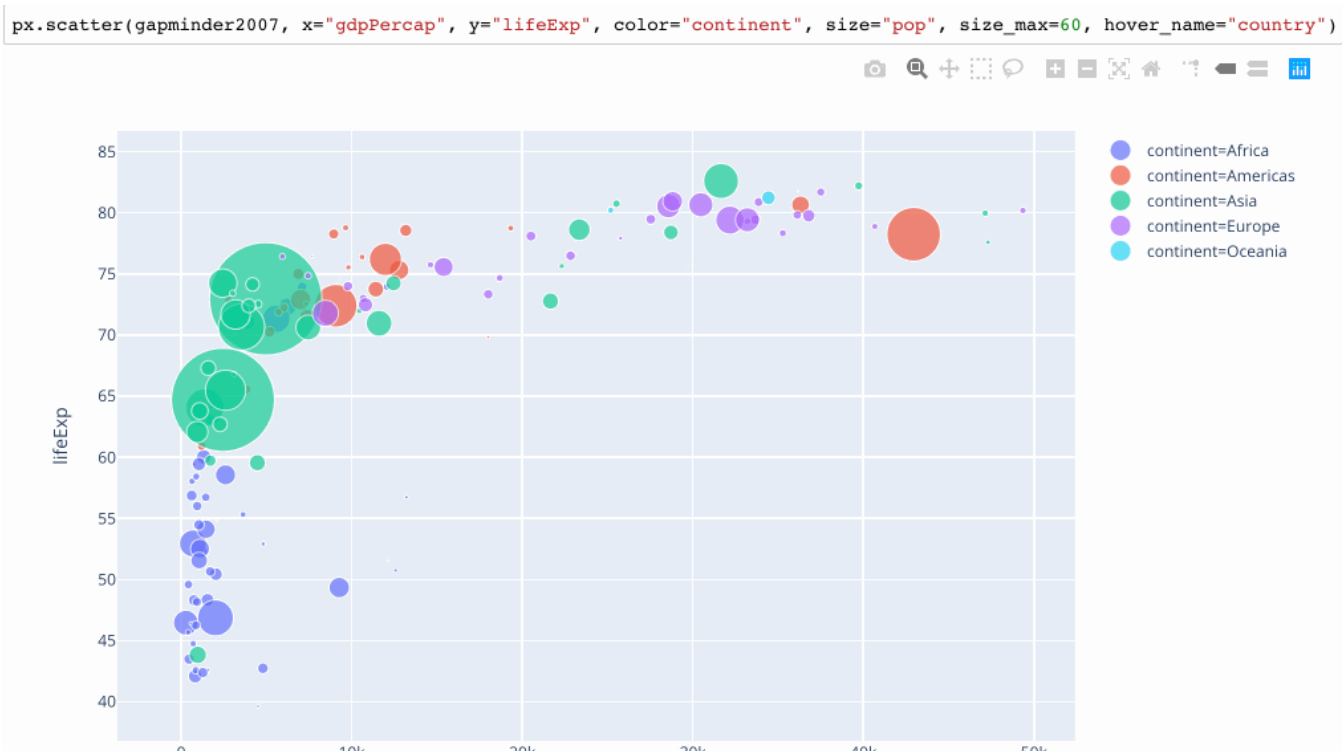
Each point here is a country, so maybe we want to scale the points by the country population… no problem: there's an arg for that too! Unsurprisingly, it's called `size`:

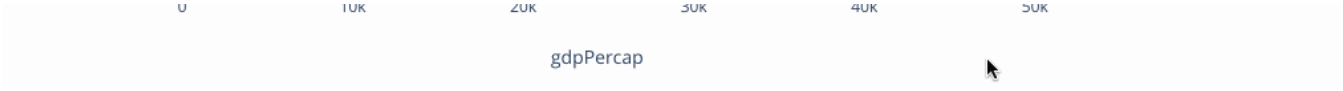
```
px.scatter(gapminder2007, x="gdpPercap", y="lifeExp", color="continent", size="pop", size_max=60)
```



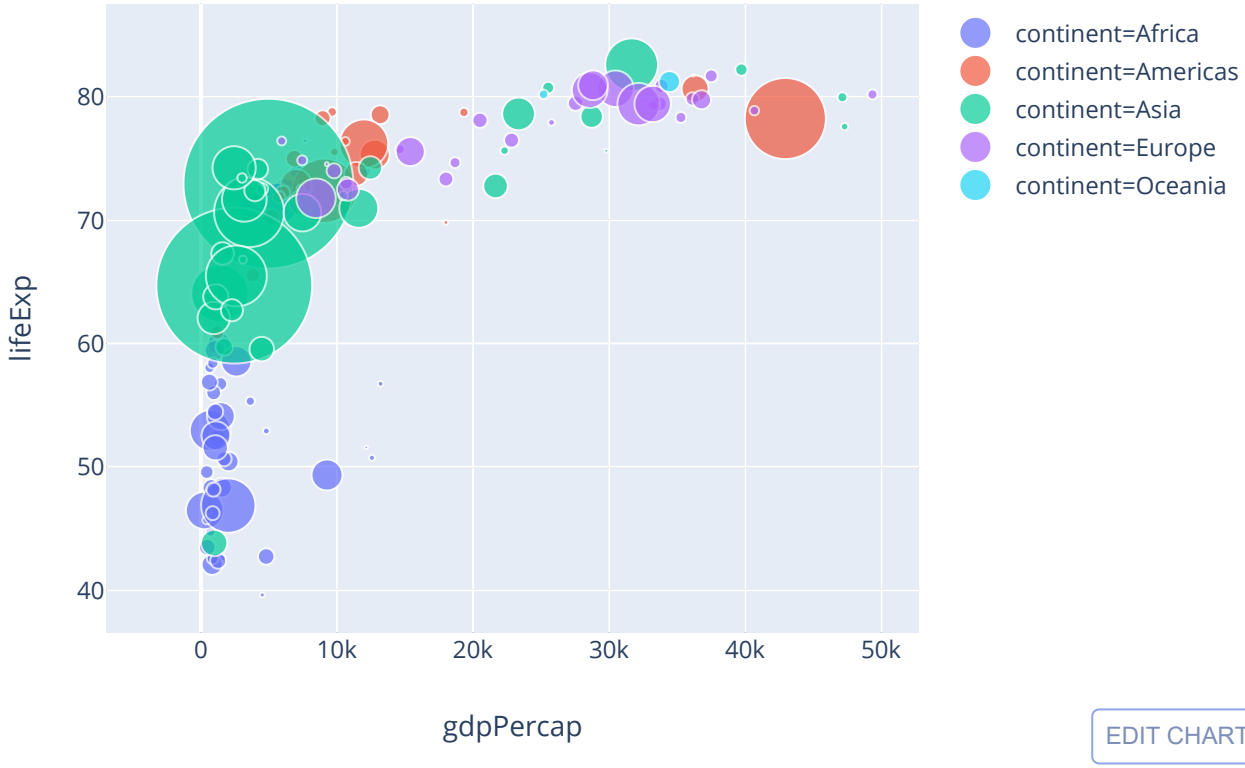
Curious about which point is which country? Add a `hover_name` and you can easily identify any point: never again wonder "what *is* that outlier?"... just mouse over the point you're interested in! In fact, the whole plot is interactive, even without `hover_name`:

```
px.scatter(gapminder2007, x="gdpPercap", y="lifeExp", color="continent", size="pop", size_max=60, hover_name="country")
```





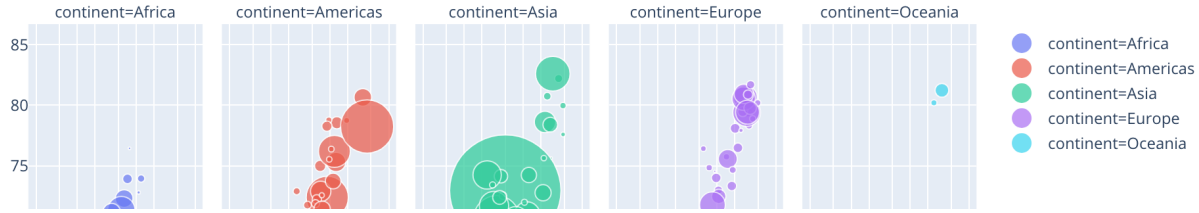
Here's an embedded version of the plot above that you can interact with right here. Try mousing over points, clicking or double-clicking on legend items, or using the "modebar" that appears when you move your mouse into the frame to control the behaviour click-drag interactions (zoom, pan, select):

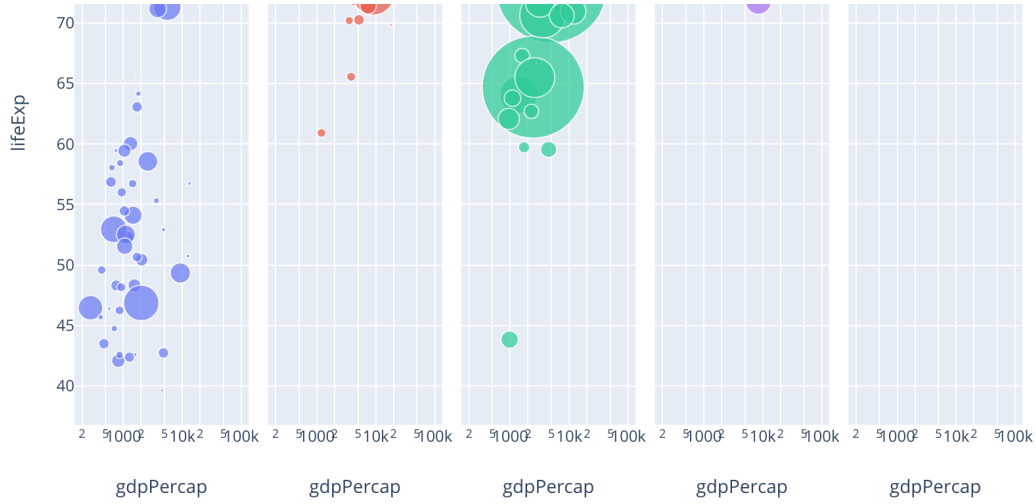


Try mousing over points, clicking or double-clicking on legend items, or using the "modebar" that appears when you move your mouse into the frame to control the behaviour click-drag interactions (zoom, pan, select).

You can also facet your plots to pick apart the continents, just as easily as coloring your points, with `facet_col="continent"`, and let's make the x-axis logarithmic to see things more clearly while we're at it:

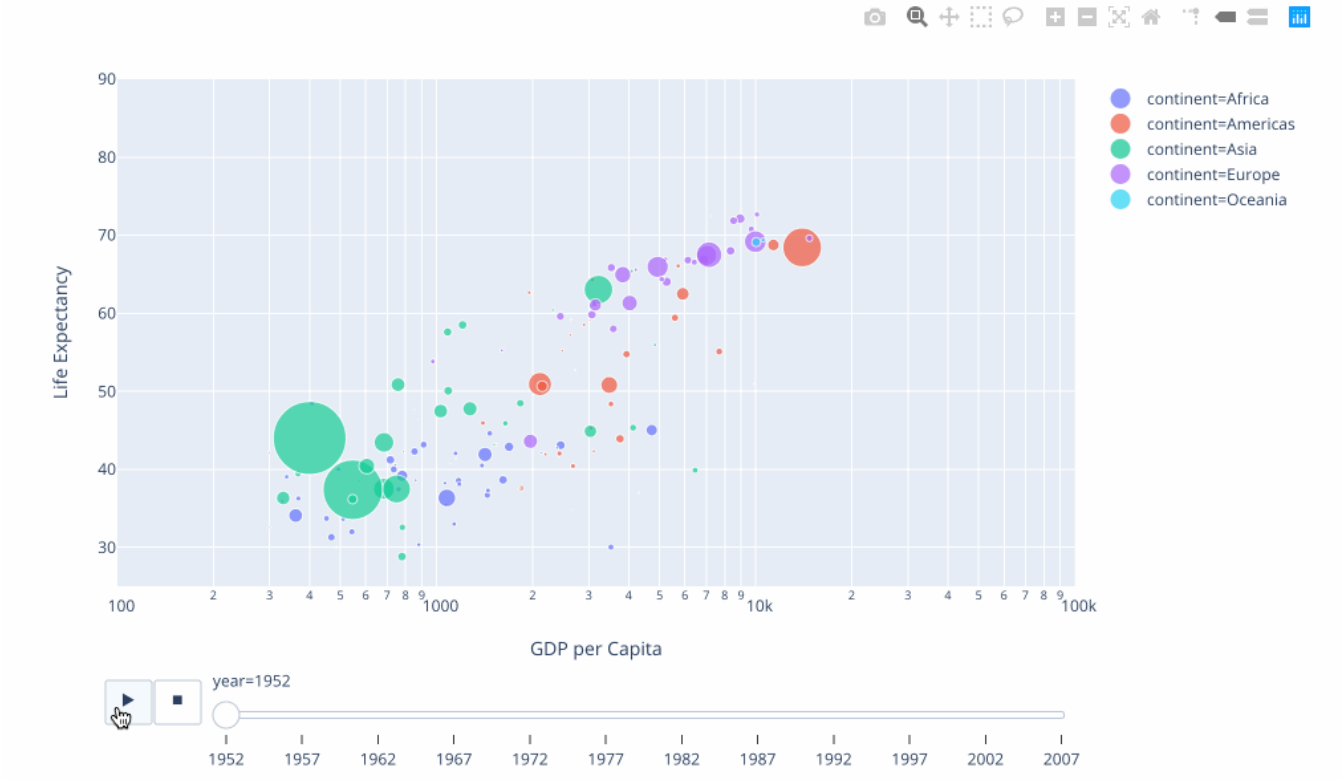
```
px.scatter(gapminder2007, x="gdpPercap", y="lifeExp", color="continent", size="pop", size_max=60,
           hover_name="country", facet_col="continent", log_x=True)
```





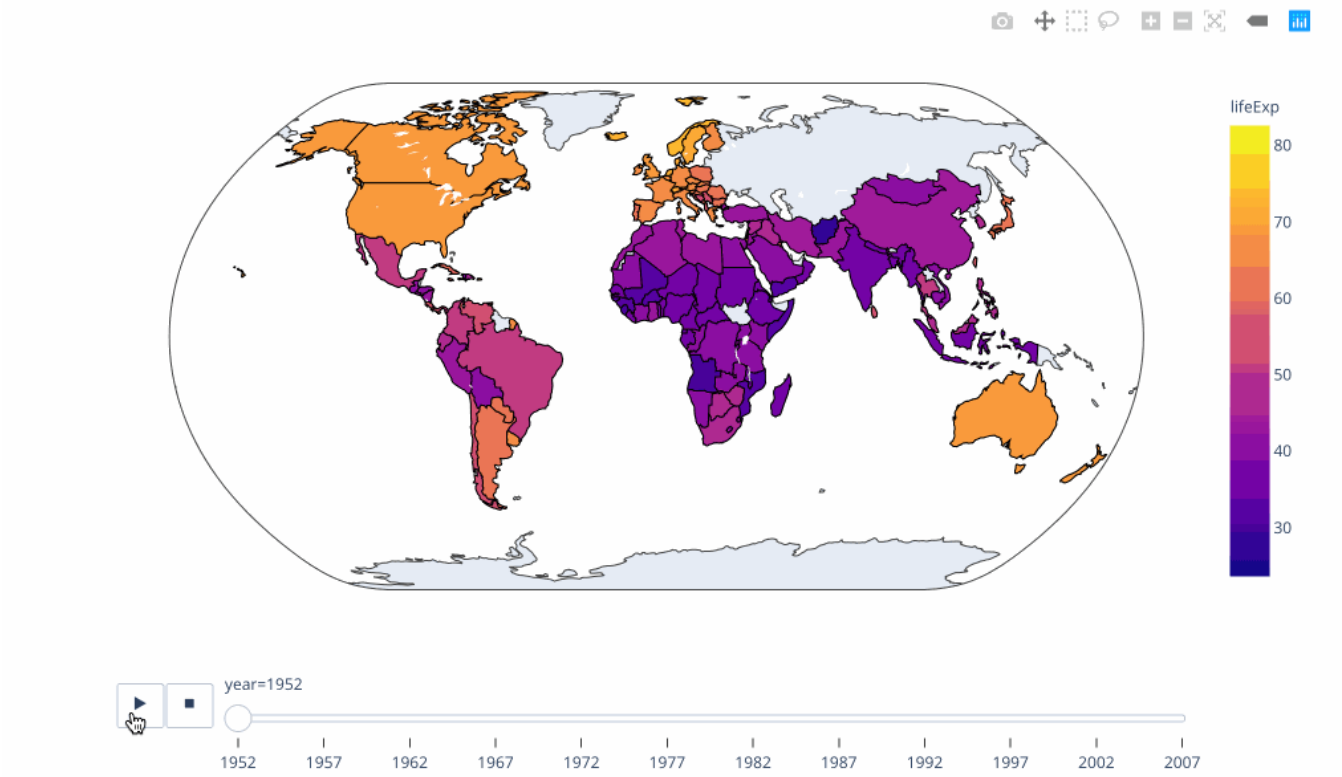
Maybe you're interested in more than just 2007 and you want to see how this chart evolved over time. You can animate it by setting `animation_frame="year"` (and `animation_group="country"` to identify which circles match which ones across frames). In this final version, let's also tweak some of the display here, as text like "gdpPercap" is kind of ugly even though it's the name of our data frame column. We can provide prettier `labels` that get applied throughout the figure, in legends, axis titles and hovers. We can also provide some manual bounds so the animation looks nice throughout:

```
px.scatter(gapminder, x="gdpPercap", y="lifeExp",size="pop", size_max=60, color="continent", hover_name="country",
           animation_frame="year", animation_group="country", log_x=True, range_x=[100,100000], range_y=[25,90],
           labels=dict(pop="Population", gdpPercap="GDP per Capita", lifeExp="Life Expectancy"))
```



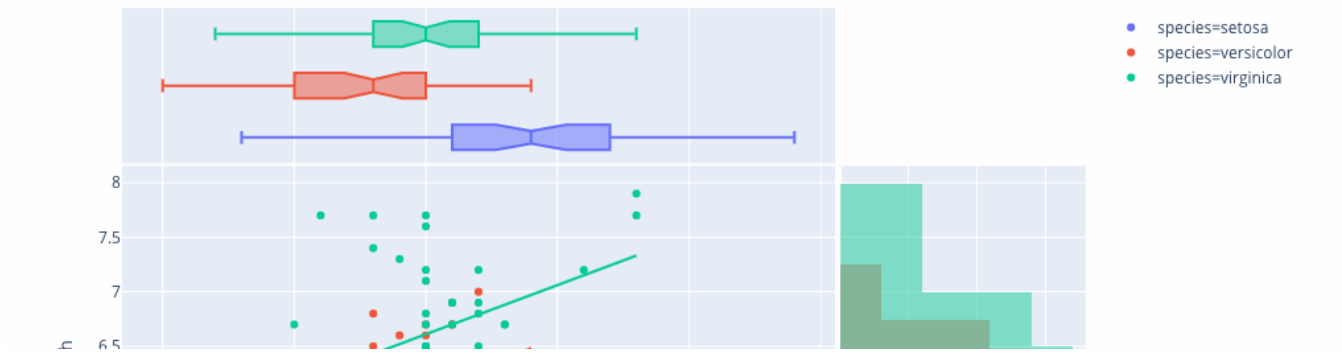
Because this is geographic data, we can also represent it as an animated map, which makes it clear that `px` can make way more than just scatter plots, and that this dataset is missing data for the former Soviet Union.

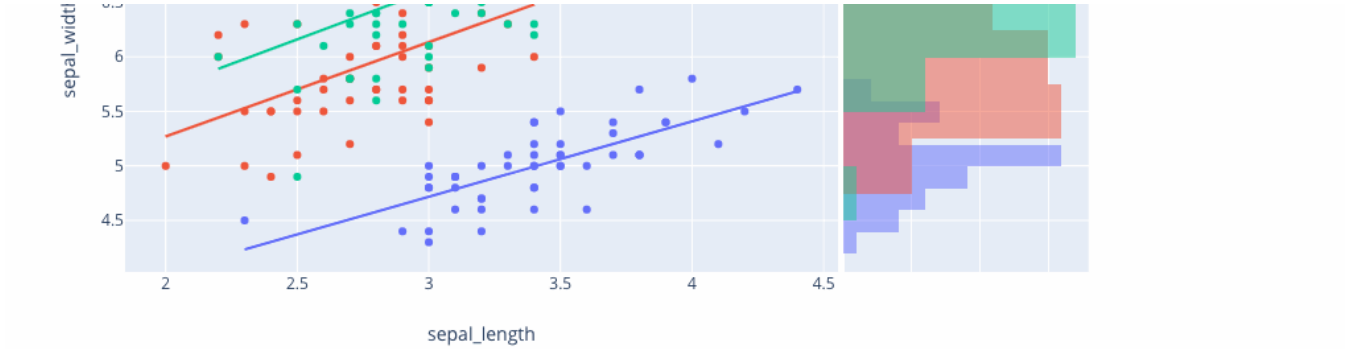
```
px.choropleth(gapminder, locations="iso_alpha", color="lifeExp", hover_name="country", animation_frame="year",
              color_continuous_scale=px.colors.sequential.Plasma, projection="natural earth")
```



In fact, Plotly Express supports **scatter** and **line** plots in 3d, polar and ternary coordinates, as well as in 2d coordinates and on maps. **Bar** plots are available in both 2d cartesian and polar flavours, and to visualize distributions, you can use **histograms** and **box or violin plots** in univariate settings, or **density contours** for bivariate distributions. Most 2d cartesian plots accept continuous or categorical data, and automatically handles date/time data as well. Check out our gallery for examples of each of these charts and the one-liners that made them!

```
px.scatter(iris, x="sepal_width", y="sepal_length", color="species", marginal_y="histogram",
           marginal_x="box", trendline="ols")
```



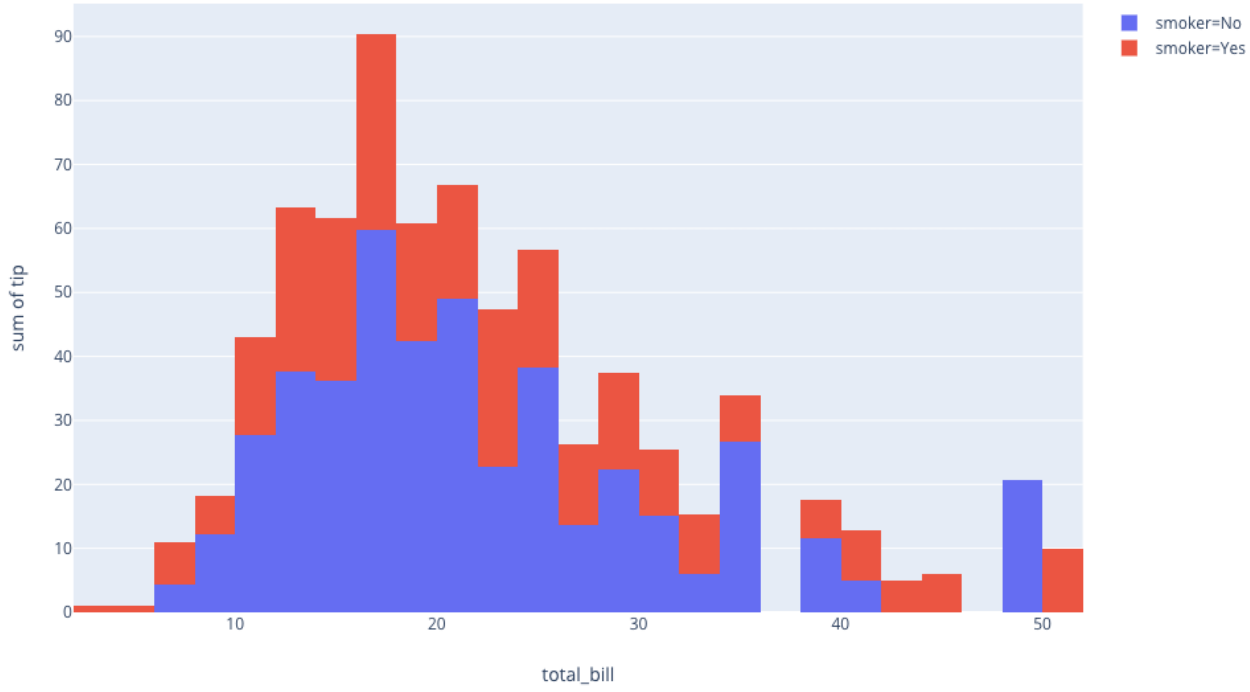


Visualize Distributions

A major part of data exploration is understanding the distribution of values in a dataset, and how those distributions relate to each other. Plotly Express includes a number of functions to do just that.

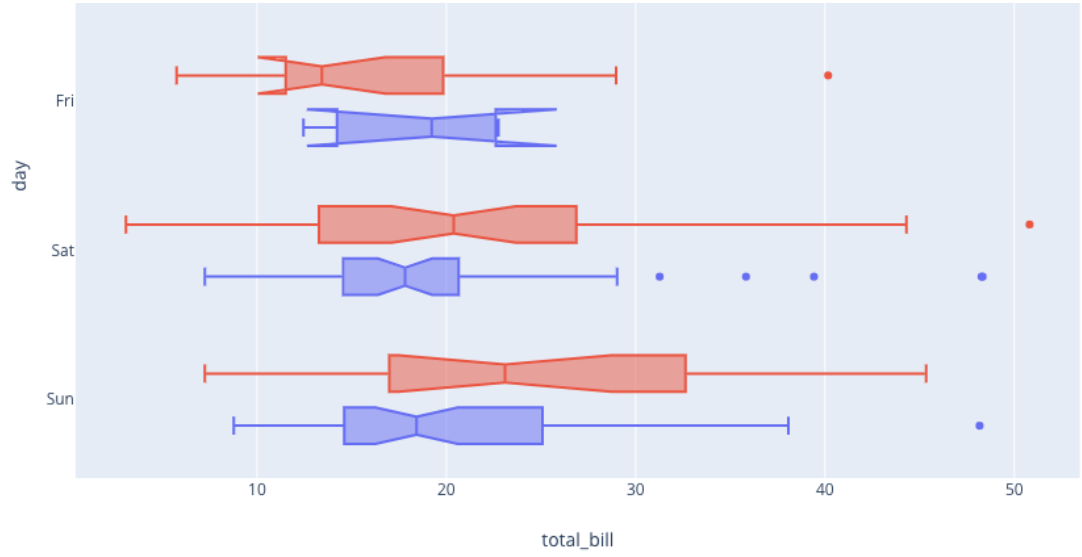
Visualize univariate distributions with histograms, box-and-whisker or violin plots:

```
px.histogram(tips, x="total_bill", y="tip", histfunc="sum", color="smoker")
```



Histograms with optional aggregation functions like 'sum' or 'average' in addition to just 'count'

```
px.box(tips, x="total_bill", y="day", orientation="h", color="smoker", notched=True,
       category_orders={"day": ["Thur", "Fri", "Sat", "Sun"]})
```



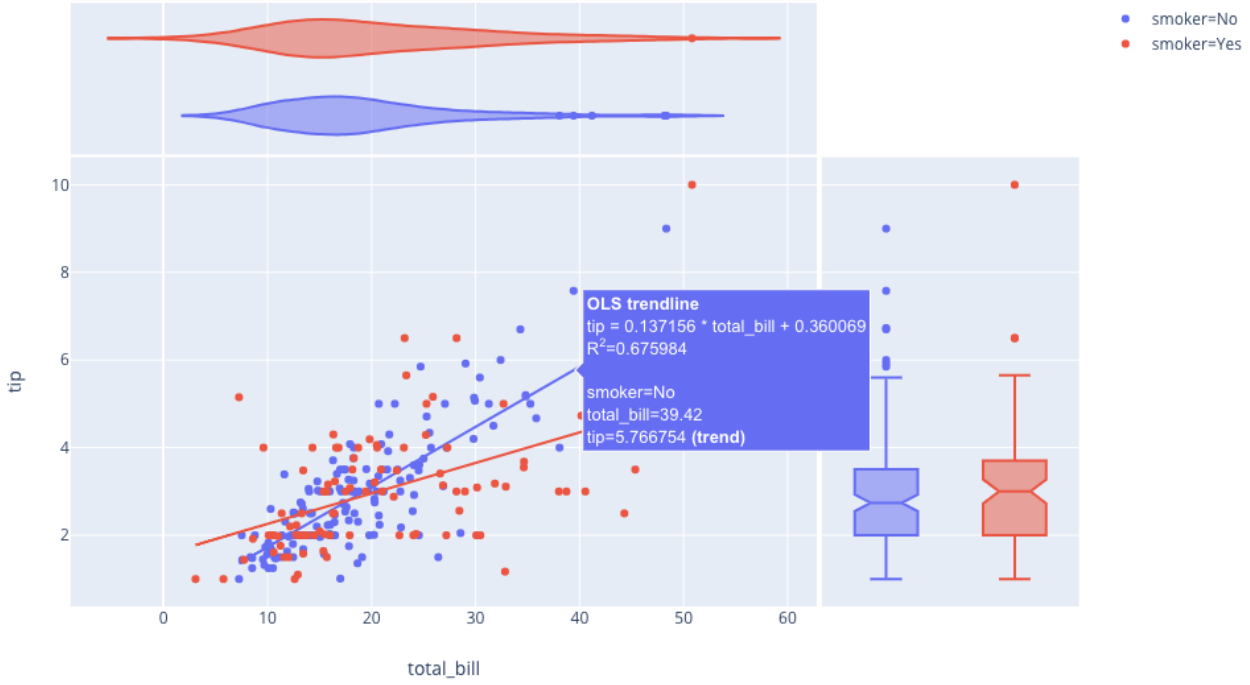
Box and whisker plots, with optional notches.

```
px.violin(tips, y="tip", x="smoker", color="sex", box=True, points="all")
```

Violin plots, with optional jittered points and embedded boxes.

You can also visualize bivariate distributions with marginal rugs, histograms, boxes or violins, and you can add trendlines too. `px` even helpfully adds the line's equation and R^2 in the hover box for you! It uses `statsmodels` under the hood to do either Ordinary Least Squares (OLS) regression or Locally Weighted Scatterplot Smoothing (LOWESS).

```
px.scatter(tips, x="total_bill", y="tip", color="smoker", trendline="ols", marginal_x="violin", marginal_y="box")
```

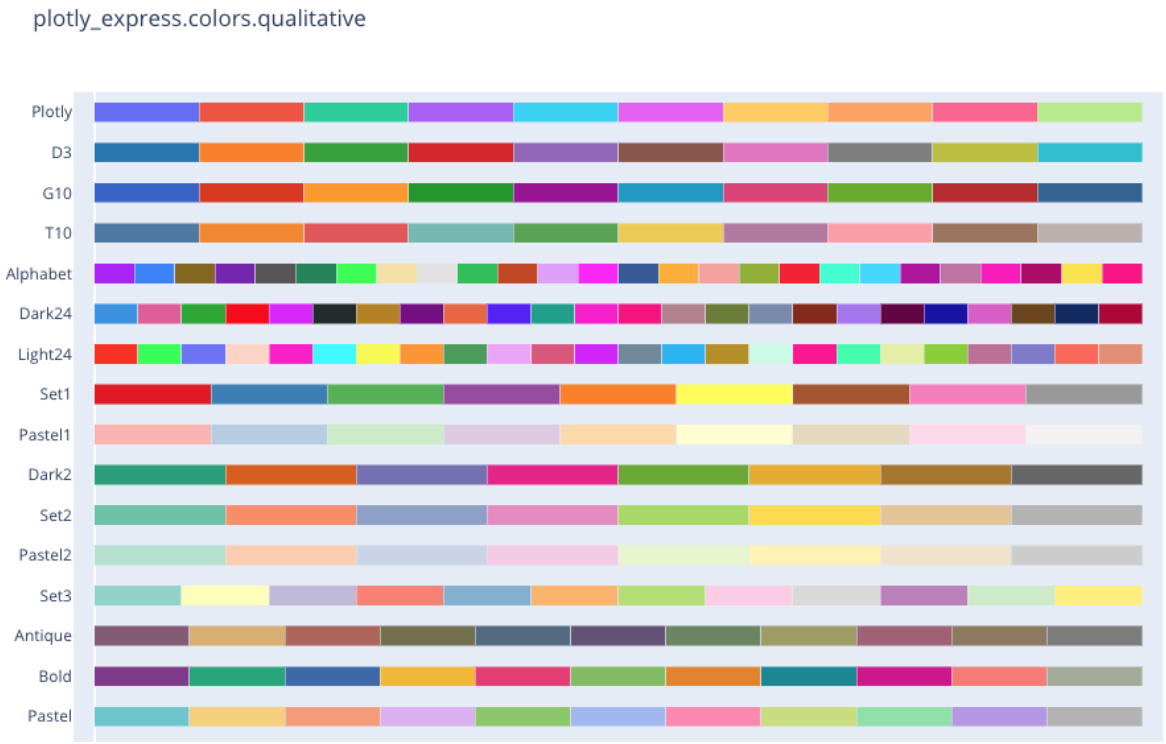


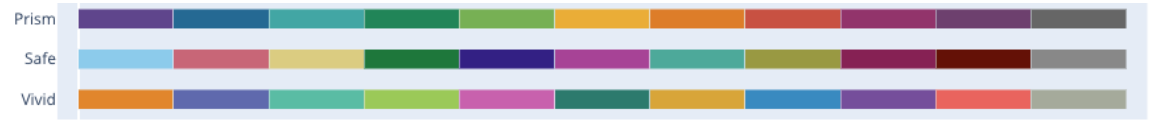
Trendlines and marginal distributions

Color scales and sequences

You'll notice some nice color scales in some of the plots above. Plotly Express. The `px.colors` module contains a number of useful scales and sequences: qualitative, sequential, diverging, cyclical, and all your favourite open-source bundles: ColorBrewer, cmocean and Carto. We've also included some functions to make browsable swatches for your enjoyment (check them out at the bottom of the gallery):

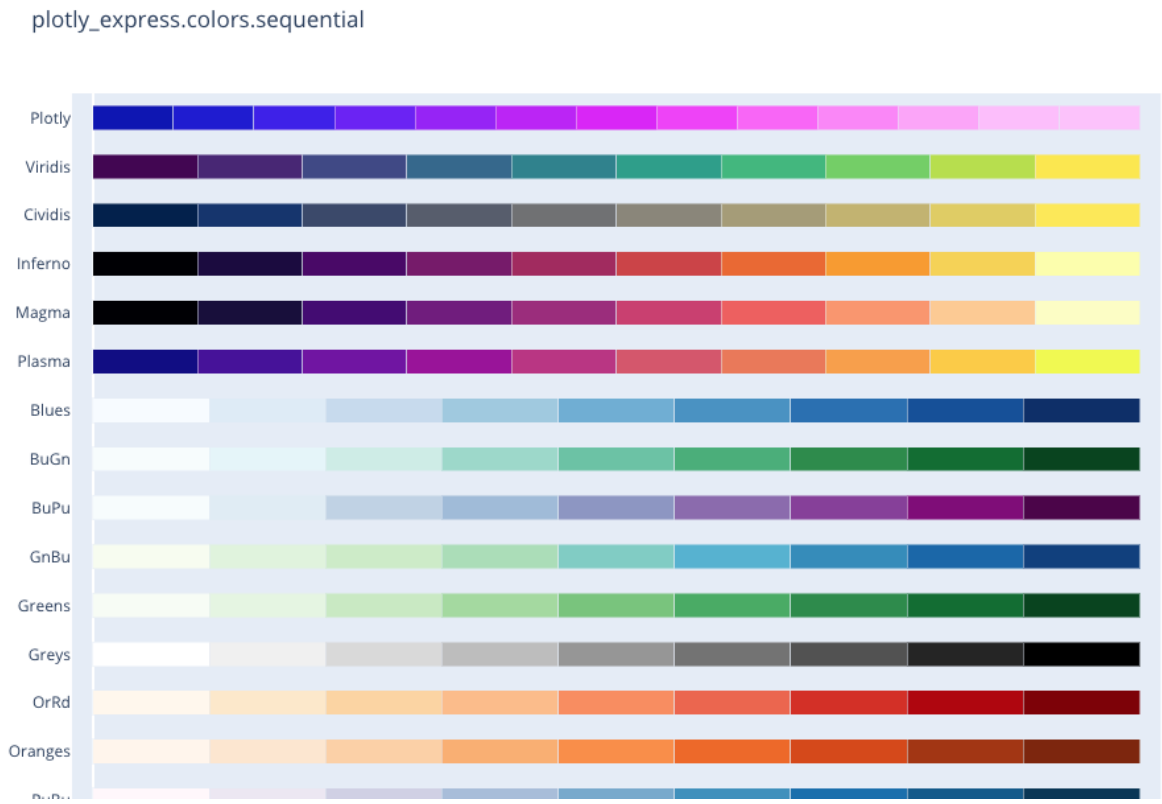
```
px.colors.qualitative.swatches()
```





Qualitative color sequences

```
px.colors.sequential.swatches()
```

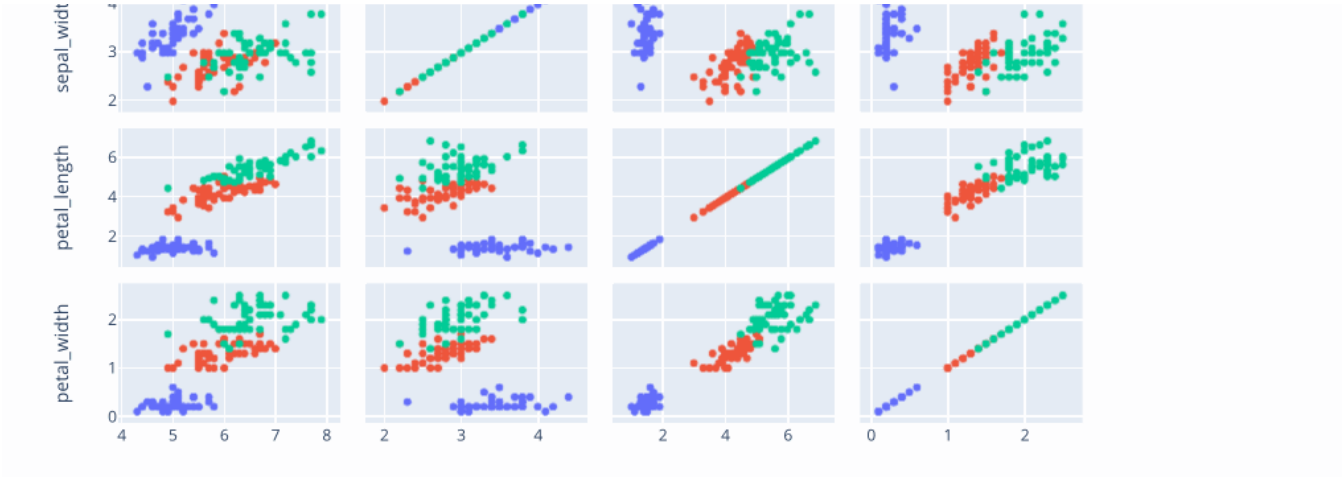


Just some of the many built-in sequential color scales

Interactive Multidimensional Visualization, in one line of Python

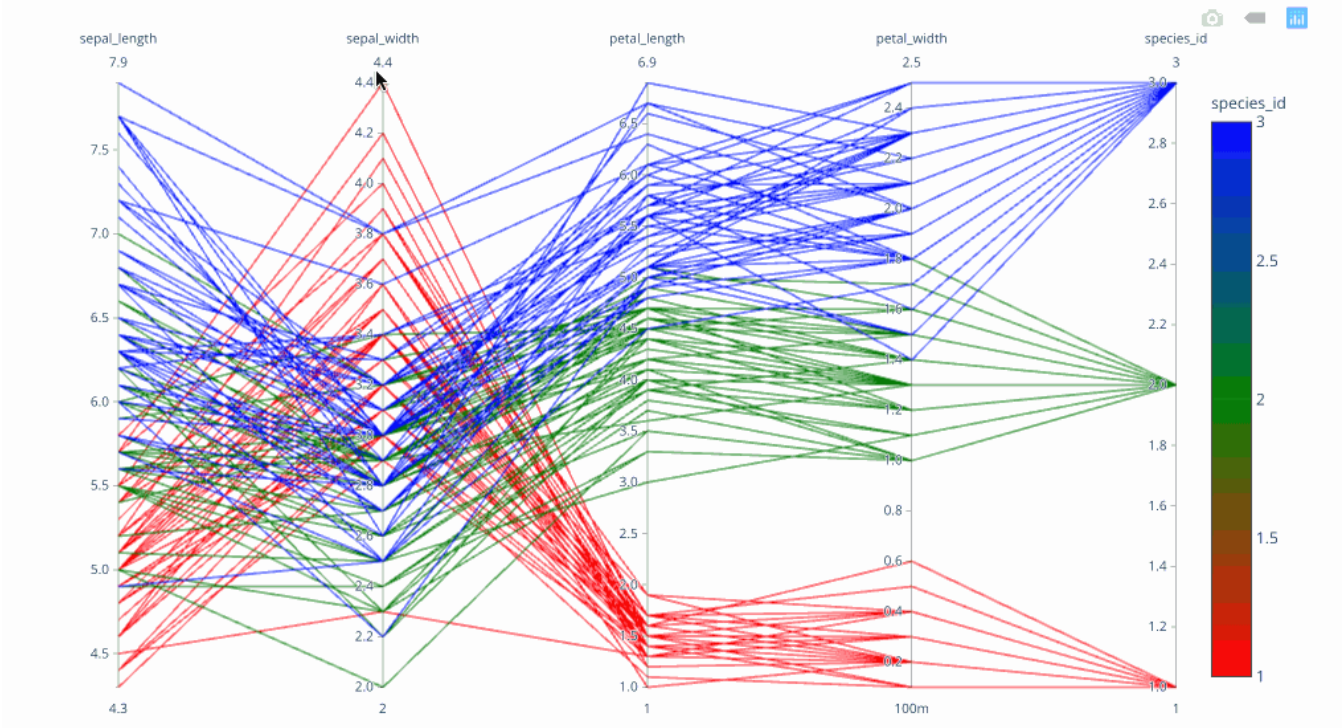
We're especially proud of our interactive multidimensional charts like scatterplot matrices (SPLOMS), parallel coordinates, and a flavour of parallel sets we call parallel categories. With these, you can visualize entire datasets in a single plot for data exploration. Check out these one-liners and the interactions they enable, right in your Jupyter notebook:

```
px.scatter_matrix(iris, dimensions=["sepal_width", "sepal_length", "petal_width", "petal_length"], color="species")
```



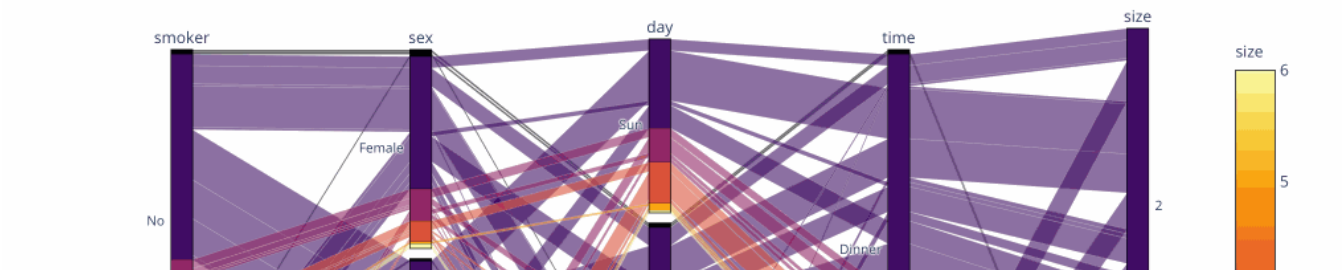
Scatterplot matrices (SPLOMs) allow you to visualize multiple linked scatterplots: every variable in your dataset vs ever other variable. Each row in your dataset appears as a point in each plot. Zoom, pan, select: all the plots are linked!

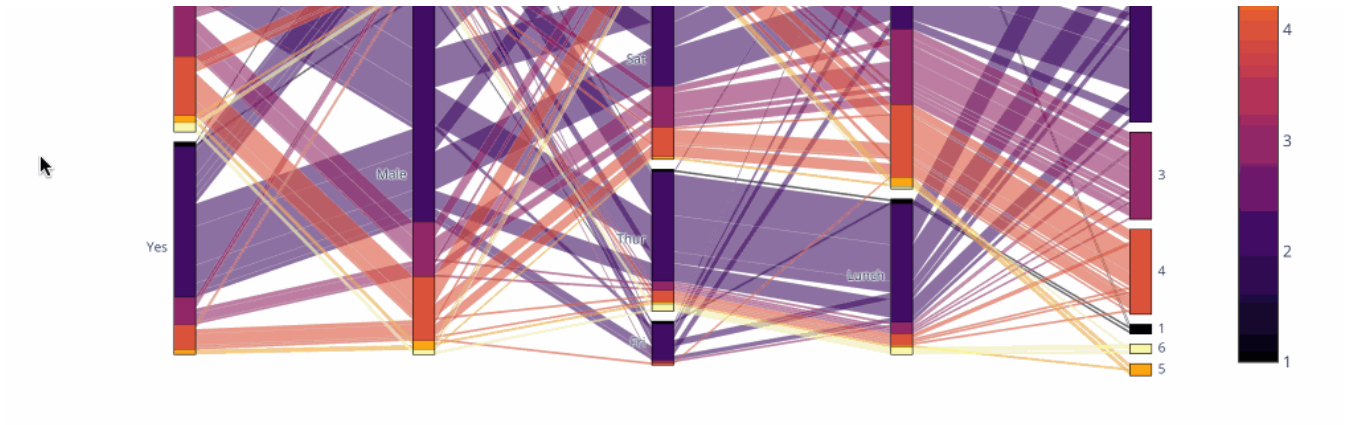
```
px.parallel_coordinates(iris, color="species_id", color_continuous_scale=["red", "green", "blue"])
```



Parallel coordinates allow you to visualize more than 3 continuous variables at once. Each row in your data frame is a line. You can drag dimensions to reorder them and select intersections between ranges of values.

```
px.parallel_categories(tips, color="size", color_continuous_scale=px.colors.sequential.Inferno)
```





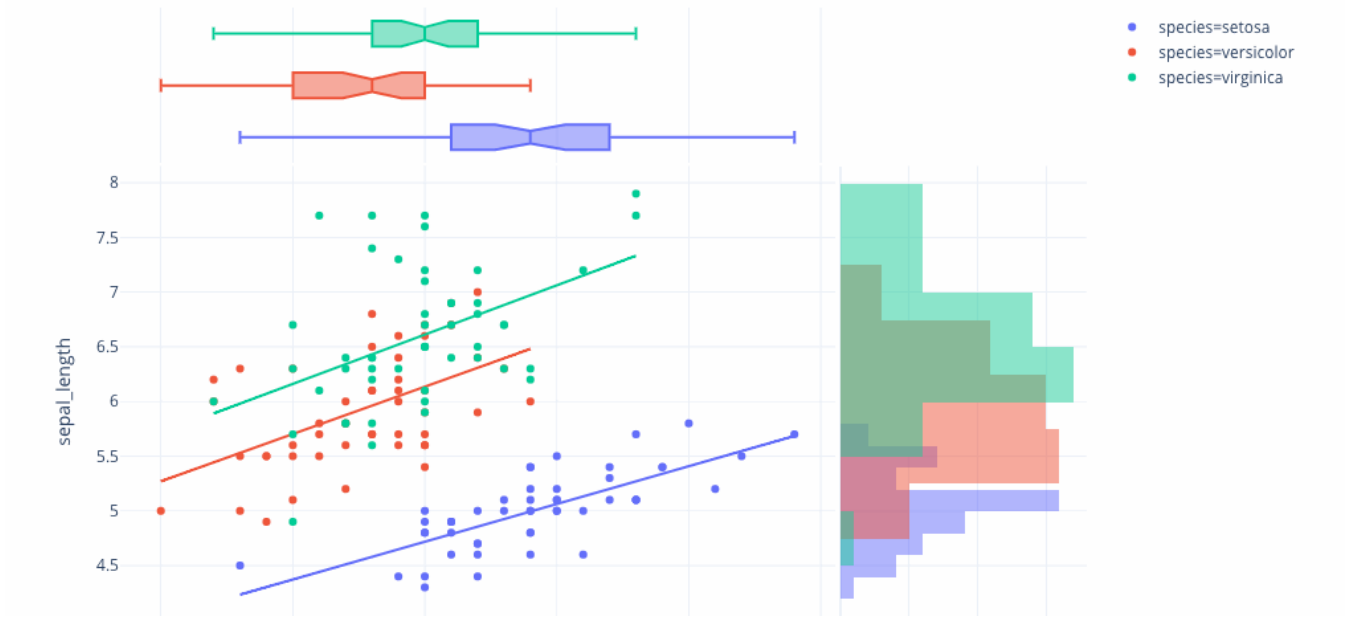
Parallel categories are a categorical analogue to parallel coordinates: use them to visualize the relationship between multiple sets of categories in your dataset.

Part of the Plotly ecosystem

Plotly Express is to Plotly.py what Seaborn is to matplotlib: a high-level wrapper that allows you to quickly create figures, and then use the power of the underlying API and ecosystem to make modifications afterwards. In the case of the Plotly ecosystem, this means that once you've created a figure with Plotly Express, you can use Themes, imperatively edit it using FigureWidgets, export it to almost any file format using Orca, or edit it in our GUI JupyterLab Chart Editor.

Themes allow you to control figure-wide settings like margins, fonts, background colors, tick positioning and more. You can apply any named theme or theme object using the `template` argument (see our Themes post for details on creating your own themes and registering their names):

```
px.scatter(iris, x="sepal_width", y="sepal_length", color="species", marginal_y="histogram", height=600,
           marginal_x="box", trendline="ols", template="plotly_white")
```



Three built-in Plotly themes: plotly, plotly_white and plotly_dark

`px` outputs objects of the class `ExpressFigure` which inherits from Plotly.py's `Figure` meaning you can use any of `Figure`'s accessors and methods to mutate a `px`-produced plot. For example, you can chain a `.update()` call to a `px` call to change legend settings and add an annotation. `.update()` now returns the modified figure so you can still do this all in one long Python statement:

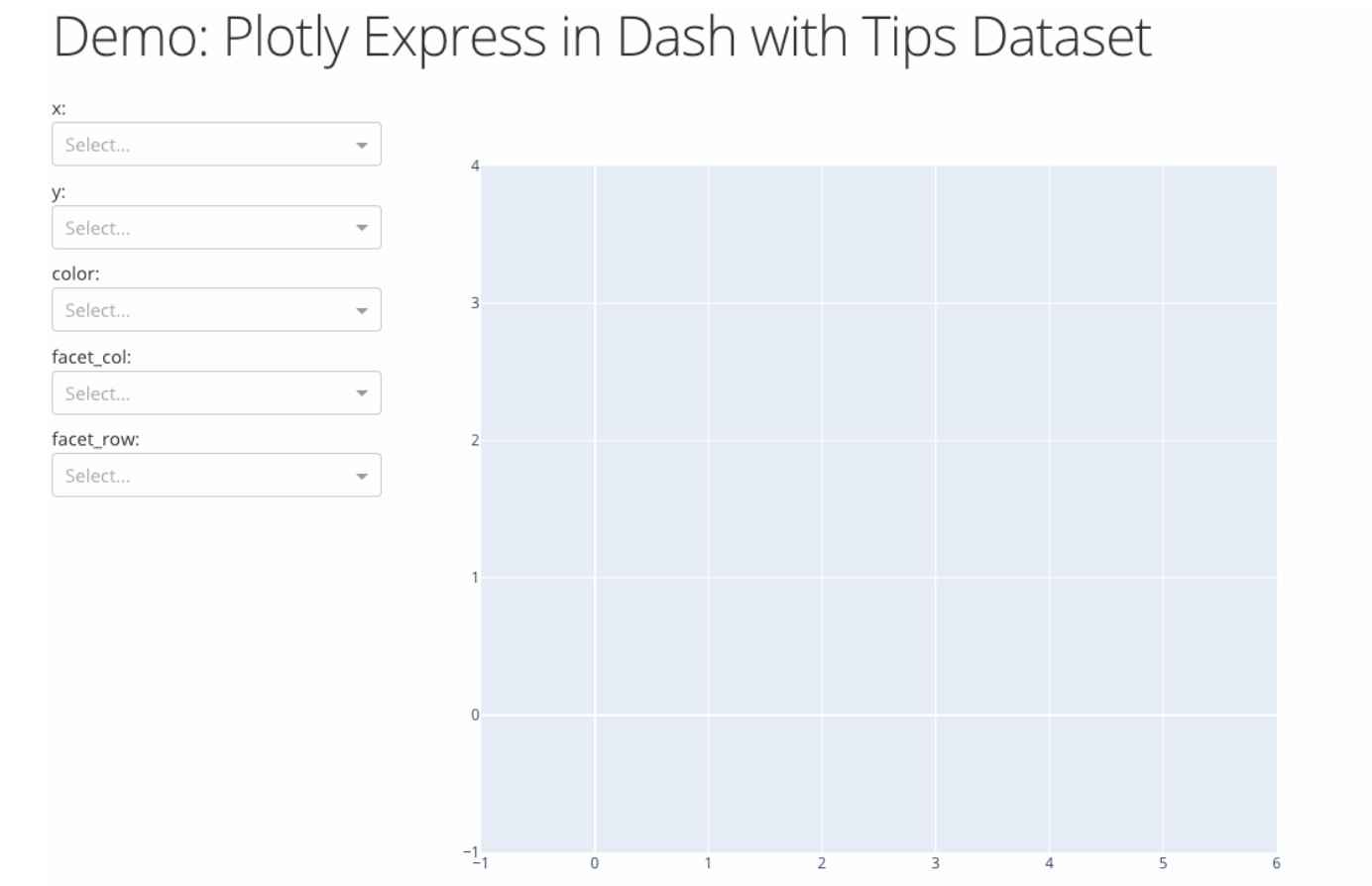
```
import plotly_express as px
fig = px.scatter(px.data.iris(), x="sepal_width", y="sepal_length", color="species")

import plotly.graph_objs as go
fig.update(layout=dict(
            legend=dict(orientation="h", y=1.1, x=0.5),
            annotations=[go.layout.Annotation(text="This one is interesting", x=3.6, y=7.2)]
           ))
```

Here we use Plotly.py's API to change some legend settings and add an annotation, after using Plotly Express to generate the original figure.

A perfect fit for Dash

Dash is Plotly's open-source framework for building analytical apps and dashboards featuring Plotly.py charts. The objects which `px` produces are 100% compatible with Dash, just pass them straight into `dash_core_components.Graph` like this: `dcc.Graph(figure=px.scatter(…))`. Here's an example of a very simple 50-line Dash app that uses `px` to generate its figures:



This 50-line Dash app uses Plotly Express to generate a UI to explore a dataset

Design Philosophy: why we built Plotly Express

There are many reasons to visualize data: sometimes you want to present some idea or result and you want to exert a lot of control over every aspect of your chart, and sometimes you want to quickly see the relationship between two variables. This is the communication-vs-exploration spectrum.

Plotly.py has grown into a very powerful tool for the communication use-case: it lets you control almost every aspect of a figure, from the placement of the legend to the length of the tick-marks. The cost of this control, unfortunately, is verbosity: it can sometimes take many lines of Python to produce figures with Plotly.py.

> *Our main goal with Plotly Express was to make it easier to use Plotly.py for exploration and rapid iteration.*

We wanted to build a library that made a different set of tradeoffs: sacrificing some measure of control early in the visualization process in exchange for a less verbose API, one that allows you to make a wide variety of figures in a single line of Python. As we showed above, however, that control isn't gone: you can still use the underlying Plotly.py API to tweak and polish the figures made with Plotly Express.

One of the main design decisions that enables such a terse API is that all `px` functions accept a "tidy" data frame as input. Every Plotly Express function embodies a crisp mapping of data frame rows to individual or grouped visual marks, and has a Grammar of Graphics-inspired signature that lets you directly map these marks' visual variables like x- or y-position, color, size, facet-column or even animation-frame to columns in your data frame. When you type `px.scatter(data, x='col1', y='col2')`, Plotly Express creates a little symbol mark for each row in your data frame – that's what `px.scatter` does — and maps the values from the column called `"col1"` to the x-position of the mark (and similarly for the y-position). The power of this approach is that it treats all visual variables the same way: you can map a data frame column to color, then change your mind and map it to size, or to a facet-row just as easily by changing the argument.

Accepting whole tidy data frames plus column names as input (as opposed to, say, raw `numpy` vectors) also allows `px` to save you a lot of keystrokes because since it knows the names of your columns, it can generate all the Plotly.py configuration to label your legend entries, axes, hover boxes, facets and even animation frames. As mentioned above, though, if your data frame columns are awkwardly-named, you can tell `px` to substitute nicer ones with the `labels` argument to every function.

The final advantage conferred by accepting only tidy input is that it supports rapid iteration more directly: you tidy your data set once, and from there on in you can create dozens of different types of figures with `px`: visualize multiple dimensions in a SPLOM, with parallel coordinates, on a map, in 2d, ternary polar or 3d coordinates, all without reshaping your data!

We haven't sacrificed all aspects of control in the name of expediency, we've just focused on the types of control you want to exert in the exploration phase of a data visualization process. You can use the `category_orders` argument to most functions to tell `px` that your categorical data "good", "better", "best" has a non-alphabetic order that matters, and it will be used in categorical axes, facet and legend orderings. You can use the `color_discrete_map` (and other `*_map` args) to pin specific colors to specific data values if that's meaningful to your use-case. And of course, you can override the `color_discrete_sequence` or `color_continuous_scale` (and other `*_sequence` args) everywhere.

At the API level, we've put a lot of work into `px` to make sure all the arguments are named so as to maximize discoverability as you type: all `scatter`-like functions start with "`scatter`" (e.g. `scatter_polar`, `scatter_ternary`) so you can discover them via

auto-completion. We opted to split these different scatter functions up so each of them would accept a tailored set of keyword arguments, particular to their coordinate system. That said, sets of functions which share a coordinate system (e.g. `scatter`, `line` & `bar`, or `scatter_polar`, `line_polar` & `bar_polar`) also have arguments which behave identically, to maximize ease of learning. We've also put a lot of effort into coming up with short and expressive names that map well onto the underlying Plotly.py attributes, to ease the transition into communication-oriented figure tweaking later in your workflow.

Finally, we should note: Plotly Express is ready for release today, but it's not finished! We want to extend faceting to all coordinate systems, add the ability to compose various `px`-generated figures together, complete our coverage of Plotly.py trace-types and more! Once we're done, Plotly Express will get rolled in to Plotly.py version 4 (when it comes out this summer) as `plotly.express`, although it will remain available as `plotly_express` as well, so don't hesitate to start using it as a standalone library today!

Getting Started

To use Plotly Express right now, just `pip install plotly_express` and head on over to our documentation pages for some copy-paste-able examples. Feel free to star and watch our Github repo to get notified of new releases. Speaking of the Github repo, if you have feedback on this library, find a bug or just want some help, please open an issue and we'll try to help you out!

We're really excited to see what scientists, analysts and engineers the world over create with `px` so feel free to share your graphics with us on Twitter: we're @plotlygraphs

Thanks

Thanks to the Plotly.js and Plotly.py teams for strong foundations to build upon, to Nicolas Kruchten for leading this effort, and to Fernando Pérez for early motivation and inspiration!

Data Visualization