Member-only story

QUICK SUCCESS DATA SCIENCE

# Introducing Seaborn Objects

One plotting ring to rule them all!

Lee Vaughan · Follow

Published in Towards Data Science · 8 min read · Mar 17, 2024


One ring to Plot them all (by Dall-E2)

Have you started using the new *Seaborn Objects System* for plotting with Python? You definitely should; it's a wonderful thing.

Introduced in late 2022, the new system is based on the *Grammar of Graphics* paradigm that powers *Tableau* and R's *ggplot2*. This makes it more flexible, modular, and intuitive. Plotting with Python has never been better.

In this *Quick Success Data Science* project, you'll get a quick start tutorial on the basics of the new system. You'll also get several useful cheat sheets compiled from the Seaborn Objects official docs.

## Installing Libraries

We'll use the following open-source libraries for this project: pandas, Matplotlib, and seaborn. You can find installation instructions in each of the previous hyperlinks. I recommend installing these in a virtual environment or, if you're an Anaconda user, in a conda environment dedicated to this project.

## The New Seaborn Objects Interface

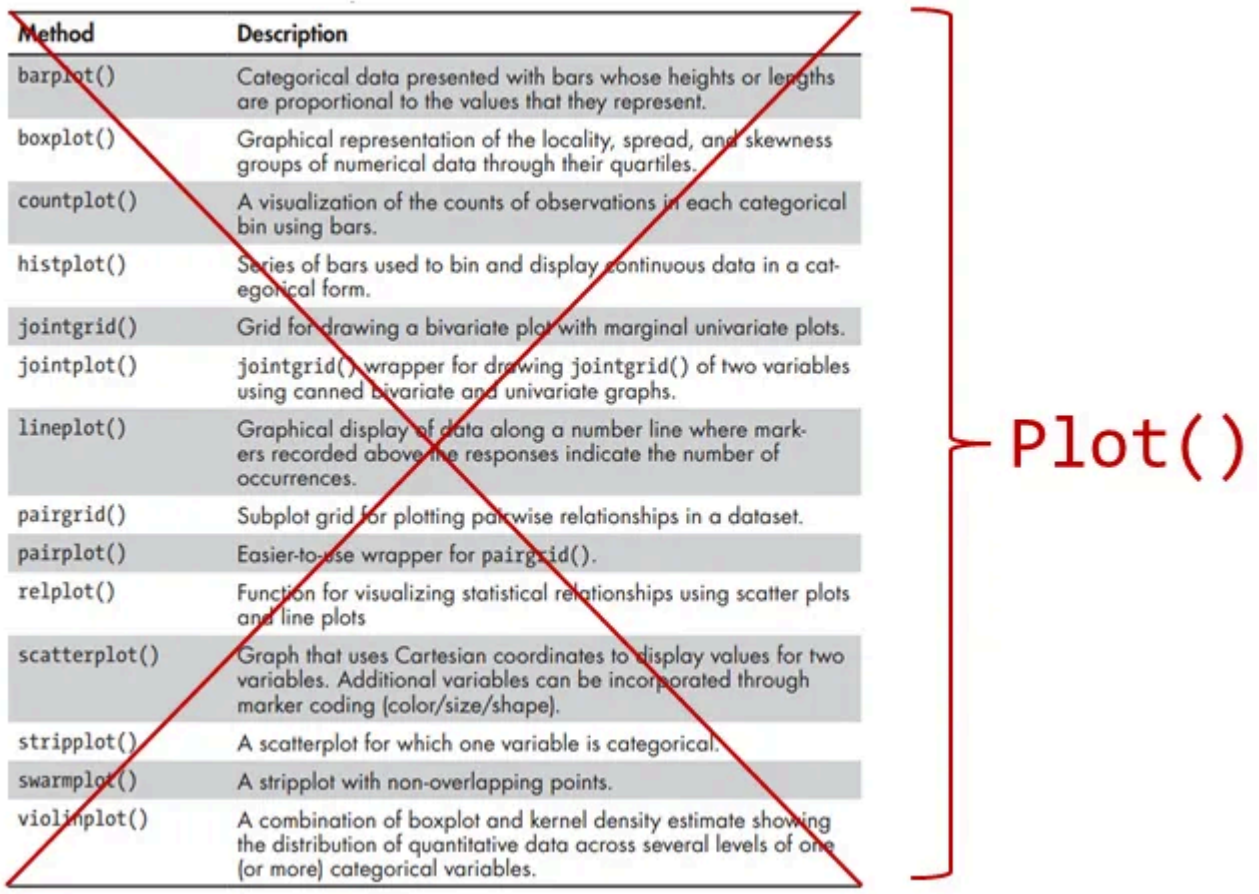
The goal of Seaborn has always been to make Matplotlib — Python's primary plotting library — both easier to use and nicer to look at. As part of this, Seaborn has relied on *declarative* plotting, where much of the plotting code is abstracted away.

The new system is designed to be even more intuitive and to rely less on difficult Matplotlib syntax. Plots are built *incrementally*, using interchangeable marker types. This reduces the number of things you need to remember while allowing for a logical, repeatable workflow.

### Everything Starts with `Plot()`

The use of a *modular* approach means you don't need to remember a dozen or more method names — like `barplot()` or `scatterplot()` — to build plots.

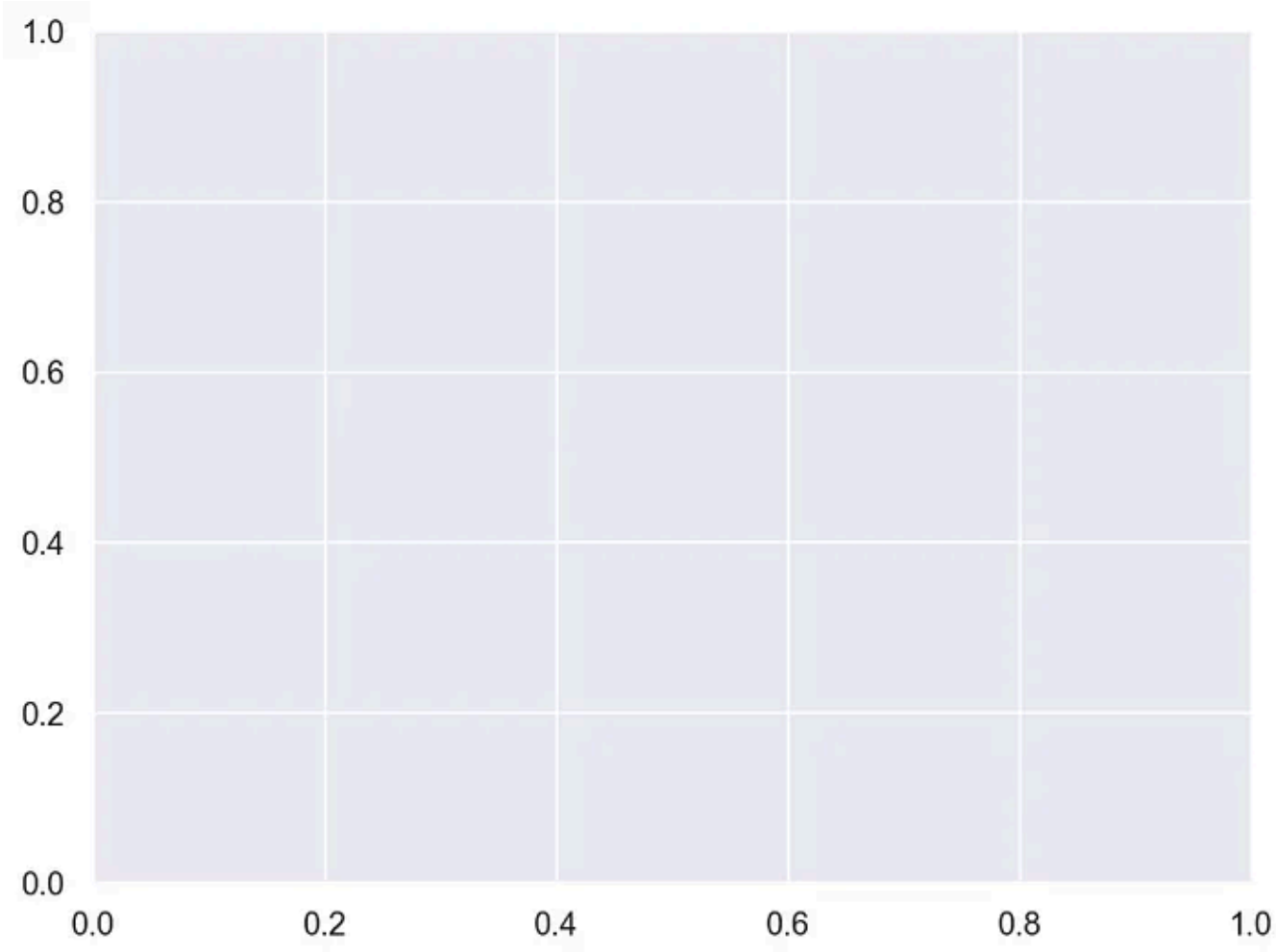
Every plot is now initiated with a single `Plot()` class.




Who needs all those old plotting methods? (by the author)

The `Plot()` class sets up the blank canvas for your graphic. Enter the following code to see an example (shown using JupyterLab):

```
import seaborn.objects as so
so.Plot()
```




Calling `Plot()` with no arguments yields an empty figure (by the author)

At this point, we don't have any data, so let's use Seaborn's built-in open-source *tips* dataset, which records restaurant data such as the total bill, the tip amount, the day of the week, the party size, and so on. Here's how to load it as a pandas DataFrame:

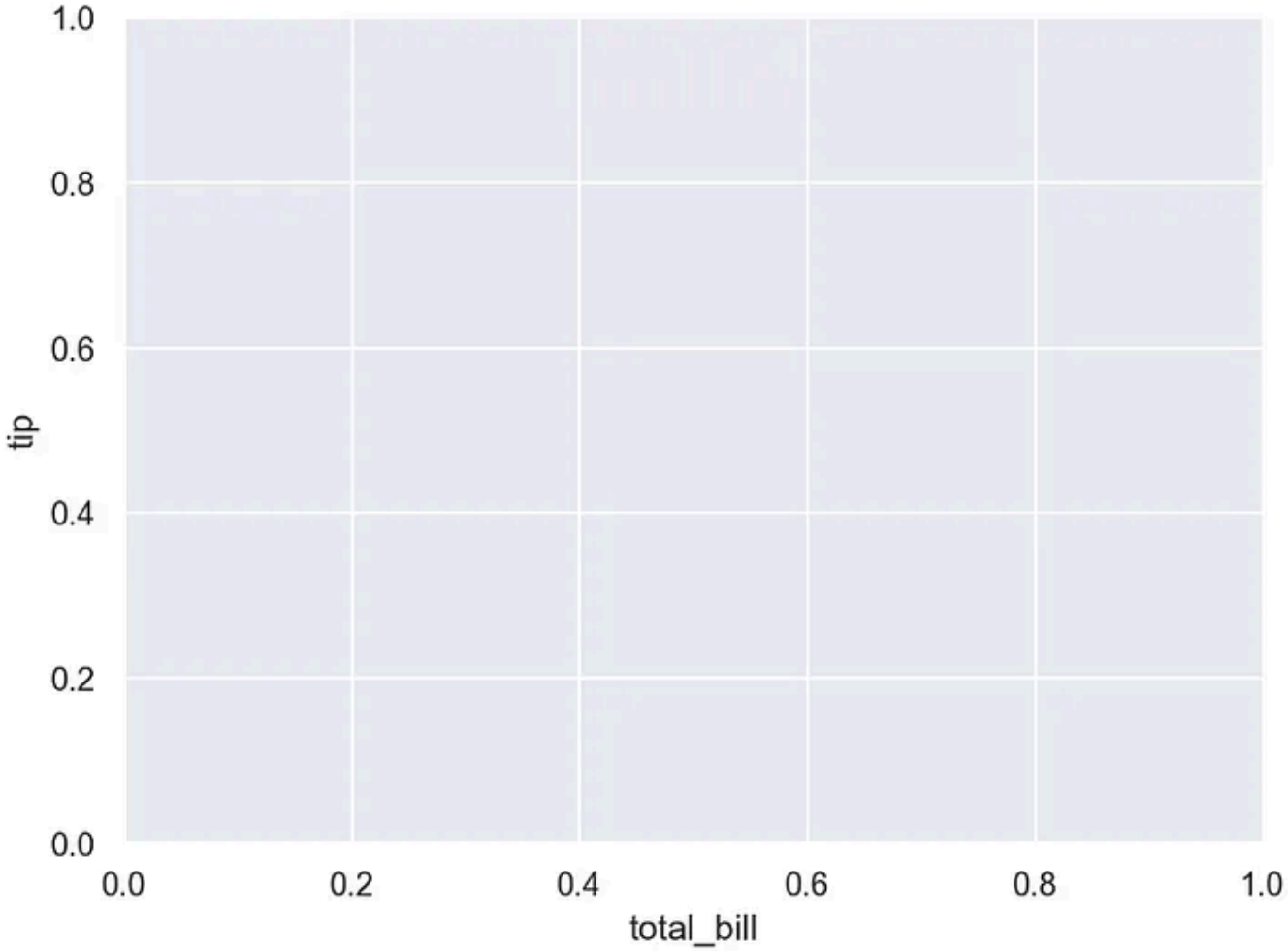
```
import pandas as pd
import seaborn as sns

# Load the tips dataset:
tips = sns.load_dataset('tips')

tips.head(3)
```

| | total_bill | tip | sex | smoker | day | time | size |
|---|---|---|---|---|---|---|---|
| **0** | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| **1** | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| **2** | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 |

The head of the tips dataframe (by the author)

Now we can point `Plot()` to the data and assign values for the x and y axes. As you might expect, the Seaborn Objects interface, like Seaborn, works very well with pandas DataFrames.

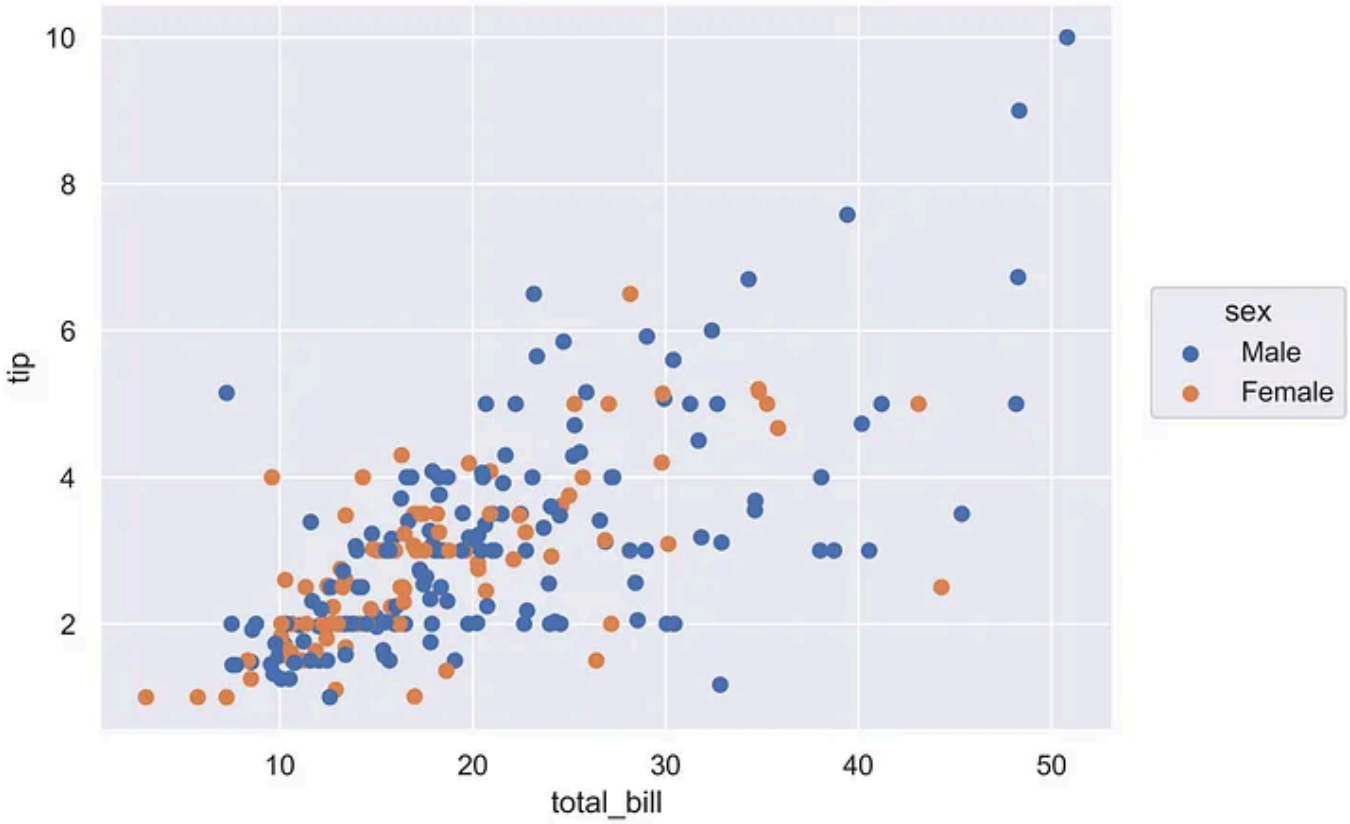
```
so.Plot(data=tips, x="total_bill", y="tip")
```




The empty figure is now aware of the dataset (by the author)

This isn't much better than the previous plot but note the x and y axes. The plot is "aware" of the underlying data. Now all we need to do is inform `Plot()` of the *type* of plot we want by specifying a `mark`. This is more intuitive than having to call a dedicated method for a plot type.

```
fig = so.Plot(data=tips, x='total_bill', y='tip').add(so.Dot(), color='sex')
# fig.show()
```




A Seaborn Objects scatterplot (by the author)

Here, we incrementally built the plot by first initiating it with a call to `Plot()` and then adding the dot markers with a call to `Dot()`. Since, intuitively, dots are used with scatterplots, this produced a scatterplot!

At this point, you've used the basic (and *required*) syntax for building a plot with the new system:

1. A call to `Plot()`,
2. Assignment of a `data` argument,
3. Assignment of a coordinate argument (such as `x` and/or `y`),
4. A call to the `add()` method to add a layer to the plot,
5. A call to a method inside `add()` to specify the marker/plot type.

Here's how it looks in its most basic form:

```
so.Plot(data=tips, x='total_bill', y='tip').add(so.Dot())
```

Note that, while you have to pass *some* mark to the `add()` method, it doesn't have to be `Dot()`. We'll look at other options shortly.

### Plot() Methods

The `Plot()` class comes with over a dozen methods that let you add marks, scale data, create faceted or paired subplots, control figure dimensions, share labels between subplots, save plots, and so on. These are summarized from the official docs in the table below.

| Plot Method | Description | Parameters |
|---|---|---|
| add() | The main method for choosing plot type. Each call adds a new layer to the plot. | mark, transforms, orient, legend, label, data, variables |
| scale() | Maps data units to visual properties. Includes transforms (log, sqrt, etc.), color palette, output ranges, scales. | *Keywords correspond to variables defined in the plot, such as coordinate variables (x, y)* |
| facet() | Creates subplots (facet grids) with conditional data subsets. | col, row, order, wrap |
| pair() | Creates subplots by pairing multiple x and/or y values. | x, y, wrap, cross |
| layout() | Controls figure size (in inches), amount of overlap, and boundaries. | size, engine, extent |
| label() | Controls labels and titles for axes, legends, and subplots. | title, legend, *variables defined in Plot() (like x, y, and color)* |
| limit() | Controls the range of visible data. | *Keywords correspond to variables defined in Plot(), and are in (min, max) tuple form* |
| share() | Controls ability to share axis ticks and limits among subplots. | *Keywords correspond to variables defined in Plot(), and use booleans or row/column* |
| theme() | Use a dictionary of rc parameters to change plot appearance. | config |
| on() | For drawing plots in existing Matplotlib figures or axes. | target *(an Axes, SubFigure, or Figure)* |
| plot() | Compiles the plot specifications and return the Plotter object. | pyplot *(boolean)* |
| save() | Compiles the plot and writes it to a buffer or disk file. | loc, *same kwargs as matplotlib.figure.Figure.savefig()* |
| show() | Compiles the plot and displays it using pyplot. Not necessary if using notebook. | *matplotlib.pyplot.show() kwargs* |

Cheat sheet for the methods of the Plot() class (by the author from the Seaborn docs)

These methods are called using dot notation. Here's the full syntax (assuming no aliases are used when importing):

```
seaborn.objects.Plot.add()
```

### Using Parentheses for Readability

With Seaborn Objects, we build plots by chaining multiple methods using dot notation. This can cause the code to become a bit unreadable. To alleviate this, it's best to enclose the plotting code in *parentheses*. This way, we can call each new method on a separate line. Here's an example where we add a regression line and a title:

```
fig = (so.Plot(data=tips, x='total_bill', y='tip')
       .add(so.Dot(), color='sex')
       .add(so.Line(color='red'), so.PolyFit())
       .label(title='Tips vs. Bill'))

fig.show()
```


We layered in the regression line by invoking the `add()` method again with the `Line()` and `PolyFit()` classes. Then we used the `Plot()` class's `label()` method to add a title. By surrounding the whole plotting code with parentheses, each of these class and method calls can be displayed on a single line, making them easy to find.

> ***NOTE:*** *If you're familiar with Seaborn, then you know that the "old" system's* `regplot()` *method automatically added the 95% confidence interval band to the*

> *regression line (see* Declarative vs. Imperative Plotting*). This feature has not yet been implemented in the new system, so not everything is better!*

### Seaborn Objects System Classes

The new system includes over two dozen classes for creating plots, adding marker types, drawing error bars and ranges, adding text, aggregating values, shifting over-posted points, and more. These are summarized from the official docs in the table below.

| Class | Description | Parameters |
|---|---|---|
| Plot() | Declaratively specify plot by adding layers comprised of a Mark and optional Stat or Move. Accepts pandas DF or dictionary with columnar values. | x, y, color, alpha, fill, marker, pointsize, stroke, linewidth, linestyle, fillcolor, fillalpha, edgewidth, edgestyle, edgecolor, edgealpha, text, halign, valign, offset, fontsize, xmin, xmax, ymin, ymax, group |
| Dot() | A mark for dot plots or less-dense scatterplots. Draws large, filled dots by default. | marker, pointsize, stroke, color, alpha, fill, edgecolor, edgealpha, edgewidth, edgestyle |
| Dots() | A dot mark designed to better handle overplotting. | marker, pointsize, stroke, color, alpha, fill, fillcolor, fillalpha |
| Line() | A mark that draws a connecting line between **sorted** observations. | color, alpha, linewidth, linestyle, marker, pointsize, fillcolor, edgecolor, edgewidth |
| Lines() | Draws a connecting line between sorted observations faster than Line() but with fewer options. | color, alpha, linewidth, linestyle |
| Bar() | Draws discrete bars between a baseline and data values. | color, alpha, fill, edgecolor, edgealpha, edgewidth, edgestyle, \|width\|, \|baseline\| |
| Bars() | A faster bar mark for histograms. Bars have full width and thin edges by default. | color, alpha, fill, edgecolor, edgealpha, edgewidth, edgestyle, \|width\|, \|baseline\| |
| Area() | A mark that fills between a baseline and data values. | color, alpha, fill, edgecolor, edgealpha, edgewidth, edgestyle, \|baseline\| |
| Hist() | Bins and counts observations with option to normalize or cumulate. | stat, bins, binwidth, binrange, common_norm, common_bins, cumulative, discrete |
| Path() | A marker connecting points in the order that they appear. | color, alpha, linewidth, linestyle, marker, pointsize, fillcolor, edgecolor, edgewidth |
| Paths() | A faster but less flexible version of Path() that connects points in the order that they appear. | color, alpha, linewidth, linestyle |
| KDE() | Computes a smooth univariate kernel density estimate. | bw_adjust, bw_method, common_norm, common_grid, gridsize, cut, cumulative |
| Dash() | Draws a line segment for each datapoint centered on the value along the orientation axis. | color, alpha, linewidth, linestyle, \|width\| |
| Range() | Draws an oriented line mark between min/max values, mainly for errorbar intervals. | color, alpha, linewidth, linestyle |
| Band() | A mark representing an interval between values. | color, alpha, fill, edgecolor, edgealpha, edgewidth, edgestyle |
| Text() | A textual mark for annotating or representing data values. | text, color, alpha, fontsize, halign, valign, offset |
| Agg() | Aggregates data along a value axis using a given method. Defaults to mean. | func (str or callable) |
| Est() | Calculates a point estimate and error bar interval. Defaults to mean and 95% confidence interval. | func, errorbar, n_boot, seed |
| Count() | Counts distinct observations and displays as a bar chart. | |

| | | |
|---|---|---|
| Perc() | Replaces observations with percentile values. Default calculation is quartiles and min/max. | k, method |
| PolyFit() | Fits a polynomial of a given order and resamples data onto the predicted curve. | order, gridsize |
| Dodge() | Displaces and narrows overlapping marks along the orientation axis. | empty, gap, by |
| Jitter() | Randomly displaces marks along one or both axes to reduce overplotting. | width, x, y, seed |
| Norm() | Scales each group relative to its maximum value. | func, where, by, percent |
| Stack() | Applies a vertical shift to eliminate overlap between marks using a baseline, such as Bar. | |
| Shift() | Layers multiple overlapping marks for improved visibility. | x, y |
| Boolean() | Input data cast (scaled) to boolean values | values |
| Continuous() | Numerical scaling supporting norms and functional transforms. | values, norm, trans |
| Nominal | A categorical scale. Currently under construction… | values, order |
| Temporal | A date/time scale. Currently under construction… | values, norm |

Cheat sheet for classes of the Seaborn Objects System (by the author from the Seaborn docs)

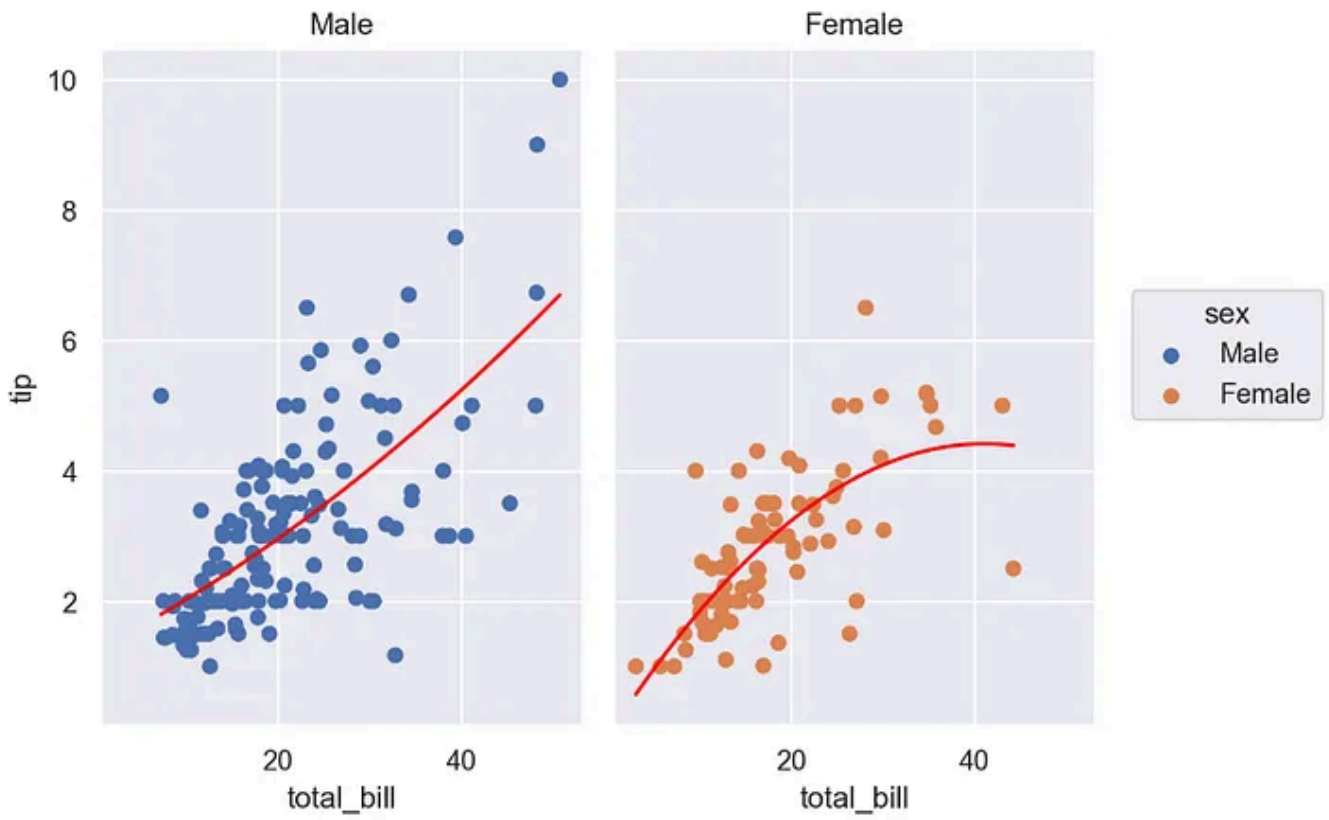
For more details and to see example plots, visit the docs. For details on styling the marker types, see "Properties of Mark Objects."

### Creating Facet Grids

The new system is excellent at preparing multi-panel plots, like facet grids and pair plots. A *facet grid* is a multi-plot grid. It allows you to subset data and compare (visualize) the subsets using columns of plots with comparable data ranges.

To create facet grids with Seaborn Objects, you use a method of the `Plot()` class, as so:

```
fig = (so.Plot(tips, 'total_bill', 'tip')
       .add(so.Dot(), color='sex')
       .add(so.Line(color='red'), so.PolyFit()))

fig.facet("sex")
```




A facet grid (by the author)

We've now "pulled apart" our previous figure so that the male and female data points aren't jumbled together. This makes it easier to see trends and limits in each conditional dataset.

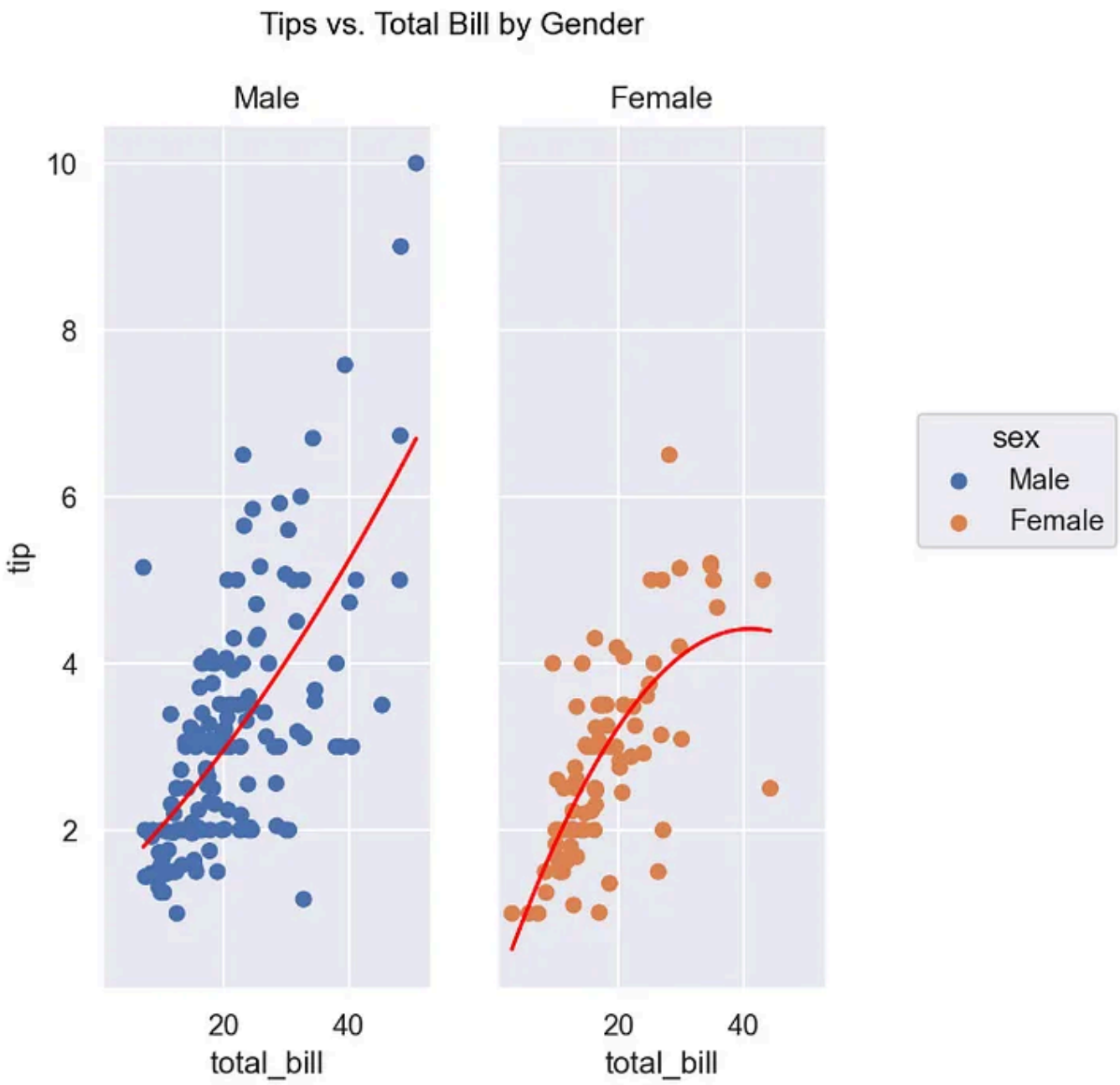
Like any declarative plotting system, Seaborn Objects is designed to abstract away much of the overhead required for making common plots, but this can result in some loss of flexibility. In addition, the system is still under

development. This means that some tasks aren't as straightforward as they are in the old system.

For example, to add a "super" title across the top of a facet grid, you have to rely on Matplotlib's `pyplot` module. First, you import it and use it to instantiate the figure ( `fig` ), then you call the old `suptitle()` method to enter the title.

After that, you use the new `Plot()` class's `on(fig)` method to post the data "on" the figure. Here's how it looks:

```
import matplotlib.pyplot as plt

fig = plt.Figure(figsize=(5, 6))
fig.suptitle("Tips vs. Total Bill by Gender")

(
    so.Plot(tips, 'total_bill', 'tip')
            .add(so.Dot(), color='sex')
            .add(so.Line(color='red'), so.PolyFit())
            .facet(col='sex')
            .on(fig)
)
```




The facet grid with a super title (by the author)

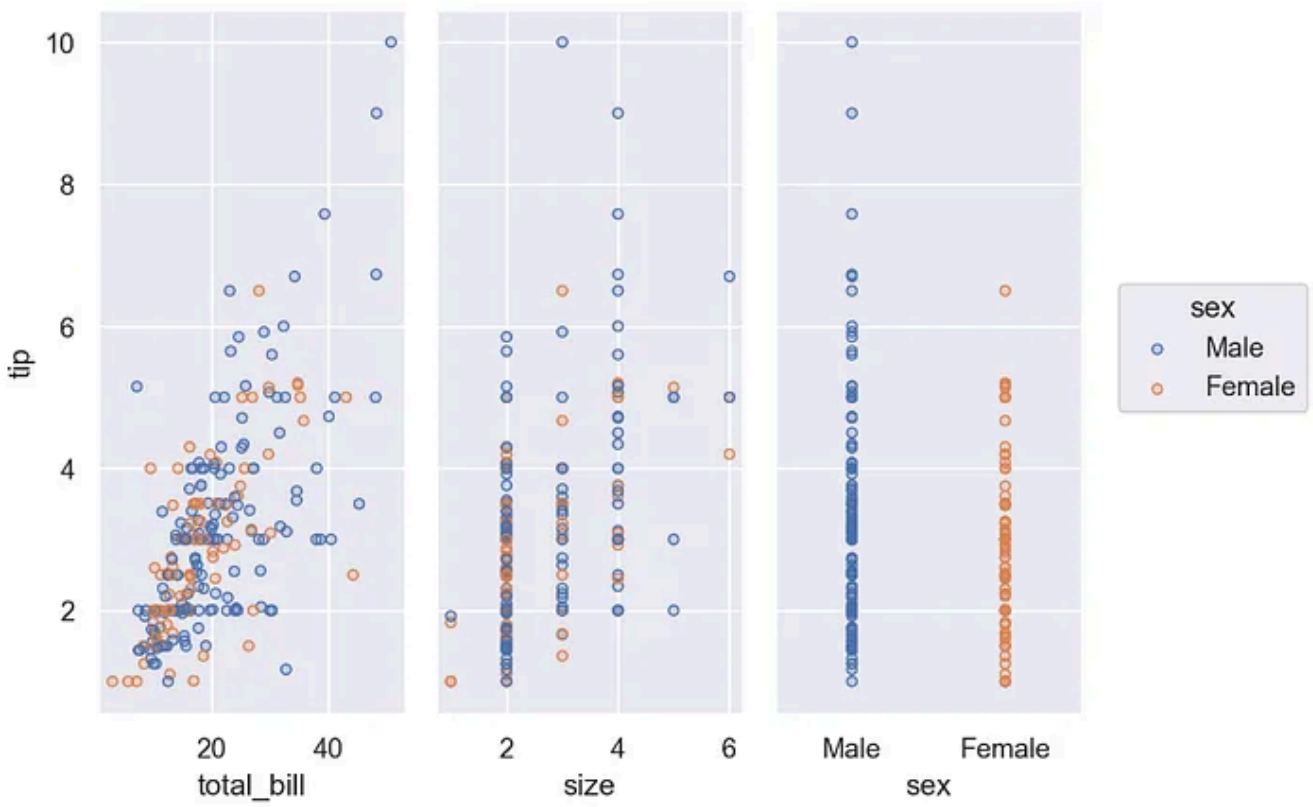
## Creating Pair Plots

A *pair plot*, also called a *scatterplot matrix*, is a data visualization technique for comparing *pairwise relationships* between multiple variables in a dataset.

To make a pair plot with Seaborn Objects, you need to specify the dataset and values for the y-axis using `Plot()`, then call the `pair()` method to choose the x-axis columns from the DataFrame, then use the `add()` method

to specify the mark type, and whether you want to color the markers by the values in a column. Here's an example:

```
(
    so.Plot(tips, y='tip')
    .pair(x=['total_bill', 'size', 'sex'])
    .add(so.Dots(), color='sex')
)
```



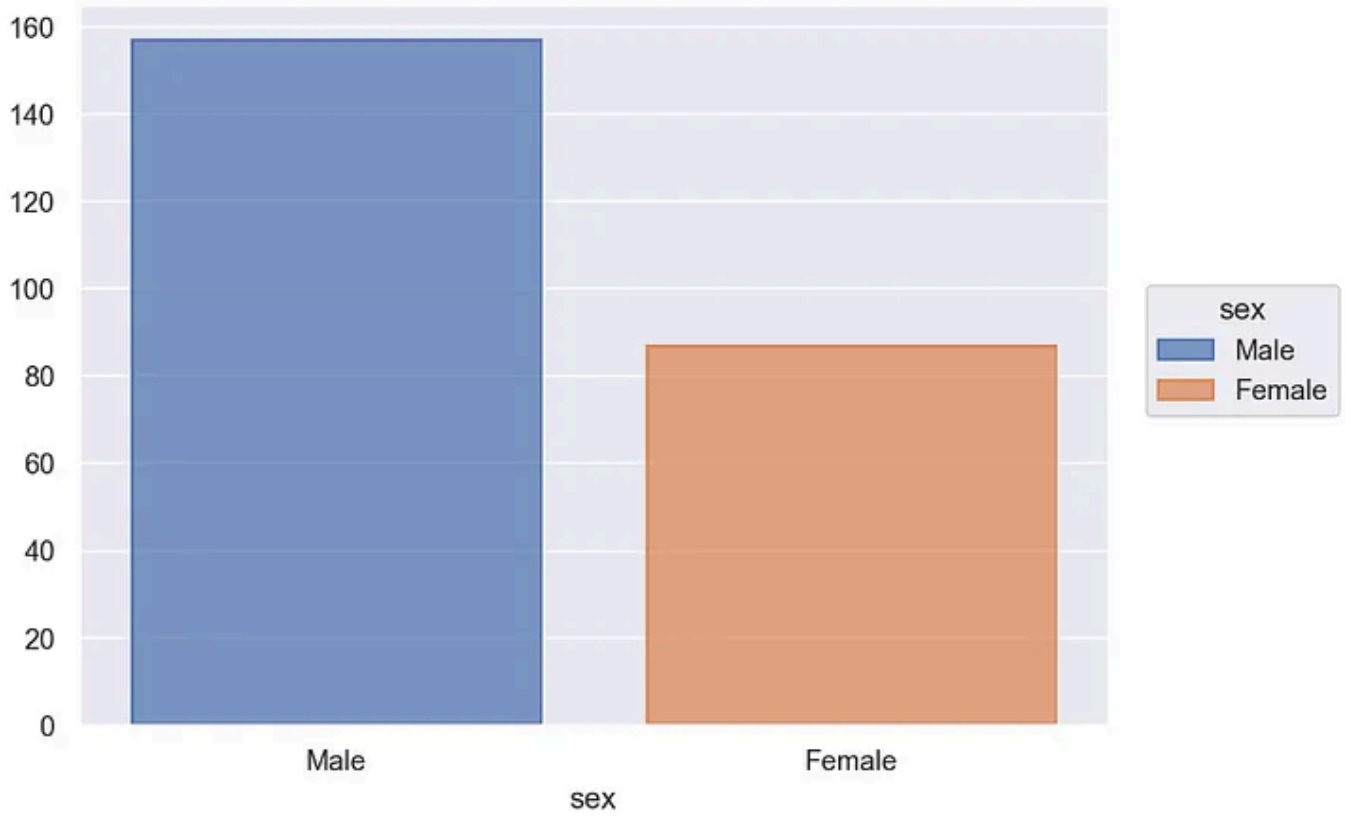

A pair plot (by the author)

Note that I use `Dots()`, rather than `Dot()`, for the markers. The goal is to improve the legibility of over-posted points by automatically using unfilled dots.

### Bars and Counts

If you were performing exploratory analysis on the tips dataset, you'd probably want to know the breakout of male vs. female customers. This is a job for the `Count()` and `Bar()` classes. Notice again how using encompassing parentheses lets you structure the code for maximum readability:

```
(
so.Plot(tips, x='sex')
    .add(so.Bar(),
         so.Count(),
         color='sex')
)
```




Bar plot (by the author)

Looks like there are almost twice as many men as women in the dataset.

## Summary

Just as Seaborn makes Matplotlib better, the Seaborn Objects System improves on Seaborn. Among the biggest changes is replacing the previous plotting methods with the `Plot()` class, which serves as the "one ring to rule them all." Every figure is now initiated using `Plot()`.

Previously, each plot type had a dedicated method, such as `sns.barplot()` for bar charts and `sns.scatterplot()` for scatterplots. To layer on more information, you often needed another, similar method, like `sns.regplot()` (for regression lines). This made it difficult to declaratively build more intricate plots.

With the modular approach of Seaborn Objects, you can now use intuitive methods, like `add()`, to layer on intuitively named markers, such as dots, lines, and bars. This "Grammar of Graphics" approach lets you build up plots using a consistent, logical methodology.

The new system is designed for making quick, "standardized" plots using declarative plotting. For more complicated plots, you'll still need to rely on imperative plotting with Matplotlib (once again, see Declarative vs. Imperative Plotting).

Finally, be aware that the new system isn't fully built out. To quote the Seaborn docs: *"The new interface is currently experimental and incomplete. It is stable enough for serious use, but there certainly are some rough edges and missing features."*

## Further Reading

Here are some additional references useful for grokking Seaborn Objects:

Anaconda: An Introduction to the Seaborn Objects System

Seaborn version 0.12.0 with ggplot2-like interface

Seaborn 0.12: An Insightful Guide to the Objects Interface and Declarative Graphics

A Quick Introduction to the Seaborn Objects System

The seaborn.objects Interface

The Grammar of Graphics

## Thanks!

Thanks for reading and please follow me for more Quick Success Data Science projects in the future.

Seaborn Objects | Seaborn Tutorial | Data Visualization | Python Programming | Grammar Of Graphics

## Written by Lee Vaughan

Follow

2.1K Followers · Writer for Towards Data Science

Author of “Python Tools for Scientists,” “Impractical Python Projects,” and “Real World Python.” Former Senior Principal Scientist for ExxonMobil.

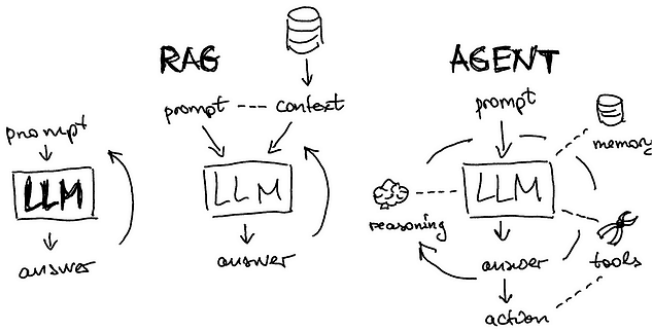
### More from Lee Vaughan and Towards Data Science

Lee Vaughan in Towards Data Science

#### Introducing Python Classes and Dataclasses

A hands-on guide for beginners

26 min read · Jan 23, 2024




Alex Honchar in Towards Data Science

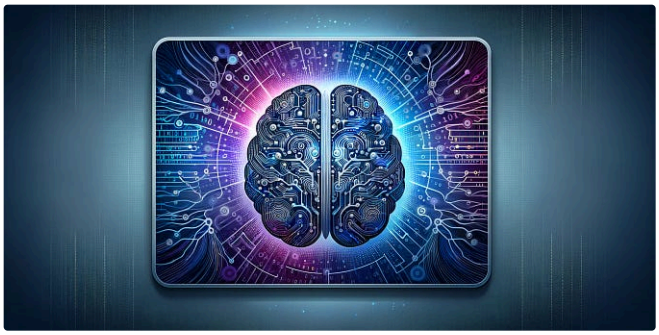
#### Intro to LLM Agents with Langchain: When RAG is Not...

First-order principles of brain structure for AI assistants

7 min read · Mar 15, 2024

957 6

1.7K 8



Cristian Leo in Towards Data Science

## The Math Behind Neural Networks

Dive into Neural Networks, the backbone of modern AI, understand its mathematics,...

28 min read · 6 days ago

1.4K 12

Lee Vaughan in Towards Data Science

## Demystifying Matplotlib

There's a reason you're confused

· 16 min read · Nov 2, 2023

1K 9

See all from Lee Vaughan

See all from Towards Data Science

## Recommended from Medium

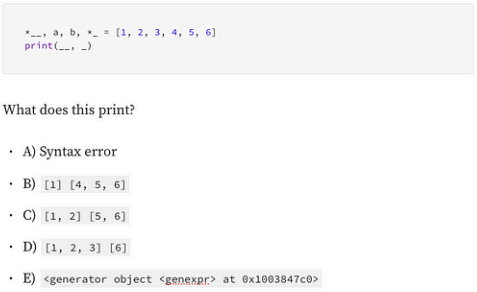
Anmol Tomar in CodeX

## 26 Python Tricks To Show Off to Your Colleagues

Tricks that will make your life easier as a Python developer

· 5 min read · Mar 25, 2024

549 2




Liu Zuo Lin

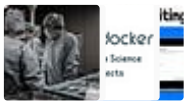
## You're Decent At Python If You Can Answer These 7 Questions...

# No cheating pls!!

· 6 min read · Mar 6, 2024

2K 14

### Lists



**Coding & Development**

11 stories · 541 saves

**ChatGPT prompts**

47 stories · 1368 saves

**Stories to Help You Grow as a Software Developer**

19 stories · 955 saves

**ChatGPT**

21 stories · 551 saves

Benedict Neo in bitgrit Data Science Publication

## Forget `pip install`, Use This Instead

Install Python packages up to 100x ⚡ faster than before.

5 min read · Mar 27, 2024

691 8

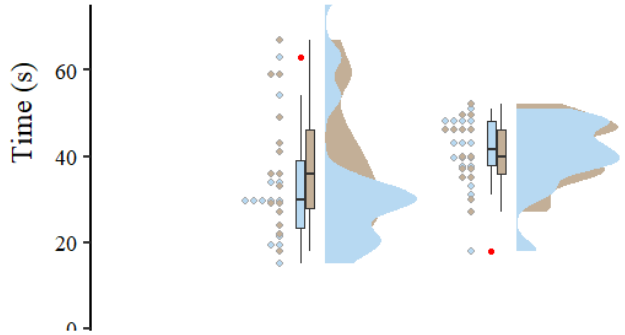
Aleksei Rozanov

## Matplotlib Makeover: 6 Python Styling Libraries for Amazing Plots

If you're a data scientist, as I am, you know that no matter how well you understand and...

5 min read · Jan 17, 2024

833 9




Daniel Manrique-Cast... in Towards Data Scien...

## Do not over-think about 'outliers', use a student-t distribution instead

The Student's t-distribution provides a robust alternative to acknowledge that our data ma...

15 min read · 5 days ago

291 4

Kallol Mazumdar in ILLUMINATION

## I Went on the Dark Web and Instantly Regretted It

Accessing the forbidden parts of the World Wide Web, only to realize the depravity of...

8 min read · Mar 13, 2024

11.1K 203

See more recommendations