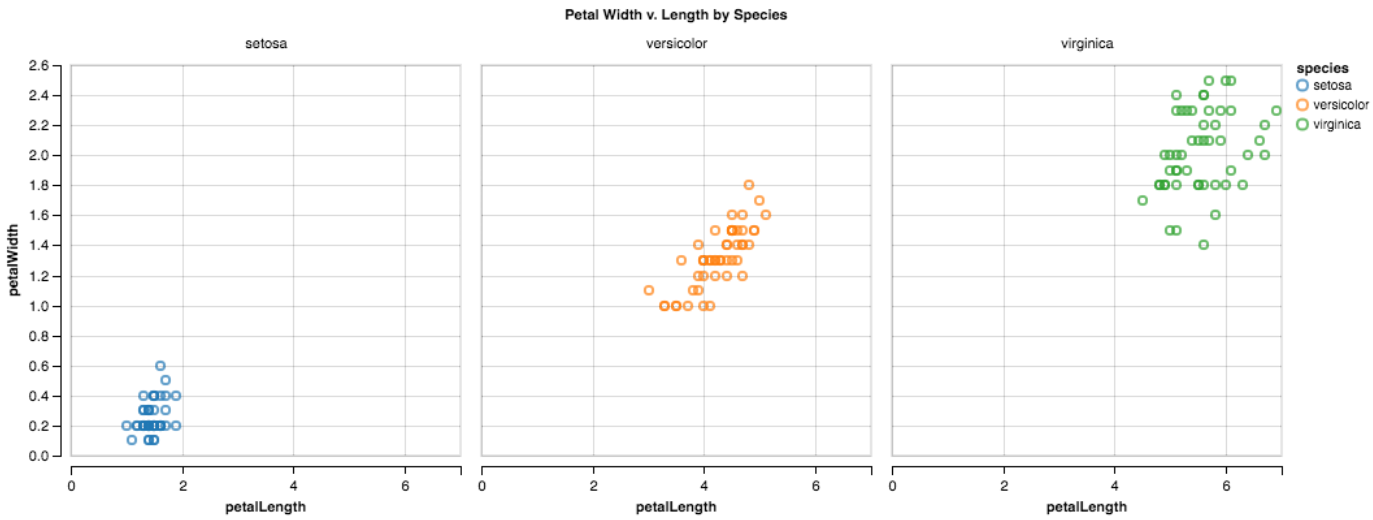
Python 数据可视化概览（涵盖 ggplot 和 Altair）

原文链接：https://github.com/xitu/gold-miner/blob/master/TODO/a-dramatic-tour-through-pythons-data…
作者用对话的形式比较了常用的 Python 可视化工具，并提供了示例代码。 —— 由cdpath分享

> - 原文地址：A Dramatic Tour through Python's Data Visualization Landscape (including ggplot and Altair)
> - 原文作者：Dan Saber
> - 译文出自：掘金翻译计划
> - 译者：cdpath
> - 校对者：Gran, Freya Yu



伙计，为什么还要去尝试？

我最近偶然发现了 Brian Granger 和 Jake VanderPlas 开发的 Altair，一个非常有潜力的新可视化库。Altair 似乎非常适合用来表达 Python 对 ggplot 的羡慕，而它采用了 JavaScript 的 Vega-Lit

我甚至是太喜欢 Altair 了，都想把本文的主题改成：**「嘿，用 Altair 吧。」**

不过我随后开始反思自以为更 Pythonic 的可视化习惯，在这相当痛苦的自我反思中，我发现自己错得一塌糊涂：为了应对手头的工作我用了一大堆工具还有乱七八糟的技术，通常是随便选一个第一个能完成工作的库 [1]。

这并不好。俗话说得好，「未经校对的绘图不值得导出 PNG 文件」。

于是我借着探索 Altair 的机会回过头来研究了 Python 可视化统计工具是如何组织在一起的。希望我的调查结果对你也有用。

我们从哪里开始呢？

本文用别出心裁的对比手法写成：「你需要做某事。你要怎么用 matplotlib，pandas，Seaborn，ggplot 或者 Altair 来实现呢？」通过这些不同的实现方式，我们就可以得出它们的优点，缺点，还有其他收获 —— 至少这么多代码说不定啥时候有用呢。

（警告：所有这一切都会以双幕剧的形式呈现）

主角们（按主观复杂性递减排列）

首先，欢迎我们的朋友们[2]:

matplotlib

matplotlib 就像八百磅的大猩猩一样「重」，最好躲着它走，除非真的需要它的力量，比如需要定制化绘图或者提供可以出版的图像。

（我们将会看到，当谈到统计可视化时，正确的思路可能是：「尽量用熟悉的工具（比如下面要讲到的四个库）把活干完，剩下的再用 matplotlib」。）

pandas

Seaborn

Seaborn 一直是我最核心的统计可视化库，它这样自我总结的：

如果说 matplotlib 试图让简单的更简单，让难的变得可行，那么 Seaborn 就是试图让虽难却定义精良的部分也变得简单。

yhat's ggplot

ggplot 是出色的声明式 ggplot2 的 Python 实现。它不仅仅「逐一复刻」了 ggplot2 的特性，还有一些共有的强大特性。（对业余的 R 语言用户而言，重要的组件似乎应有尽有。）

Altair

新成员 Altair 是「声明式统计可视化库」，有着极其好用的 API。

好极了。既然大伙都到了还做了自我介绍，我们开始尴尬的晚宴对话吧。我们的演出叫……

Python 可视化库小商店（每个库演的就是自己）

第一幕：线和点

（在第一场，我们要处理的是整洁数据集 `ts`。它有三列：`dt`（存日期），`value`（存值）和 `kind`（有四个不同的**水平**：A，B，C 和 D）。数据长这个样子：）

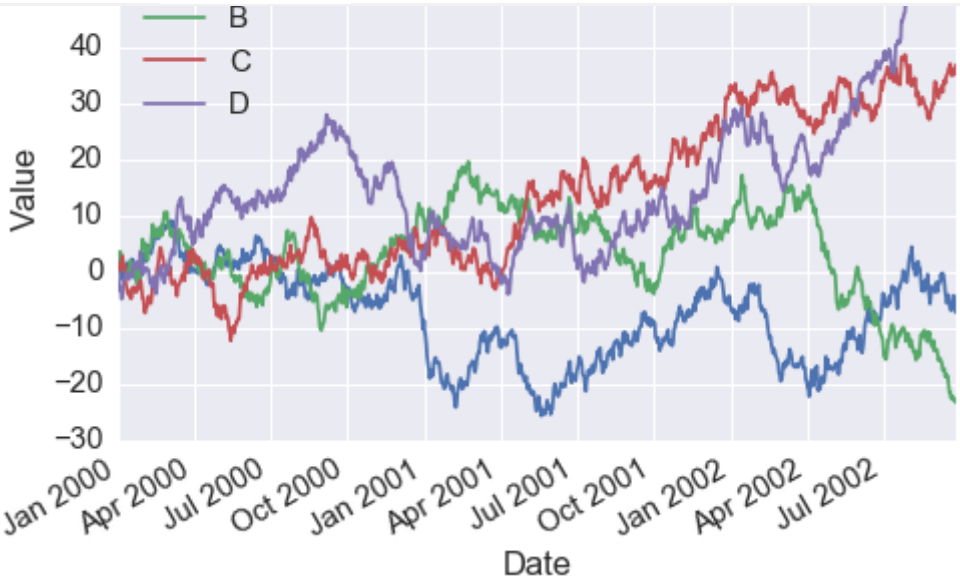
0	2000-01-01	A	1.442521
1	2000-01-02	A	1.981290
2	2000-01-03	A	1.586494
3	2000-01-04	A	1.378969
4	2000-01-05	A	-0.277937

第一场：如何在一张图上画多个时间序列？

---

matplotlib: 哈！哈哈！不能再简单了。虽然我可以用很多复杂的方式搞定这个，不过我明白你们的笨脑子是无法理解其中的精妙的。所以我退而求其次给你们展示两个简单的方法。第一个方法，我循环使用你们虚构的矩阵，我相信你们这些人把它叫做「数据框」，取其子集传给相关的时间序列。然后调用 `plot` 方法，传入子集中的相关列。

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))

for k in ts.kind.unique():
    tmp = ts[ts.kind == k]
    ax.plot(tmp.dt, tmp.value, label=k)

ax.set(xlabel='Date',
       ylabel='Value',
       title='Random Timeseries')

ax.legend(loc=2)
fig.autofmt_xdate()
```

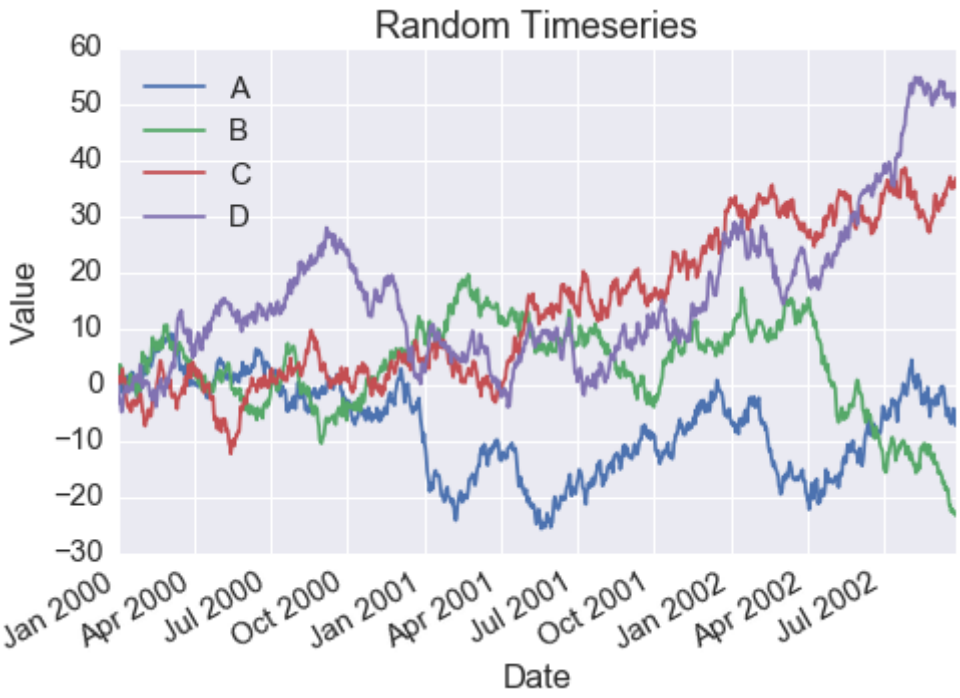


MPL: 然后我把它转换成数组（给 pandas 做手势），让他对「数据框」做轴向旋转（pivot），结果是这样的：

```
# the notion of a tidy dataframe matters not here
dfp = ts.pivot(index='dt', columns='kind', values='value')
dfp.head()
```

kind	A	B	C	D
dt				
2000-01-01	1.442521	1.808741	0.437415	0.096980
2000-01-02	1.981290	2.277020	0.706127	-1.523108
2000-01-03	1.586494	3.474392	1.358063	-3.100735
2000-01-04	1.378969	2.906132	0.262223	-2.660599
2000-01-05	-0.277937	3.489553	0.796743	-3.417402

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))

ax.plot(dfp)

ax.set(xlabel='Date',
       ylabel='Value',
       title='Random Timeseries')

ax.legend(dfp.columns, loc=2)
fig.autofmt_xdate()
```



pandas (看上去怯生生的): 这很不错，Mat。真的不错。谢谢你提到我。我也能用同样的方法搞定这个 —— 希望可以同样出色（微微一笑）。

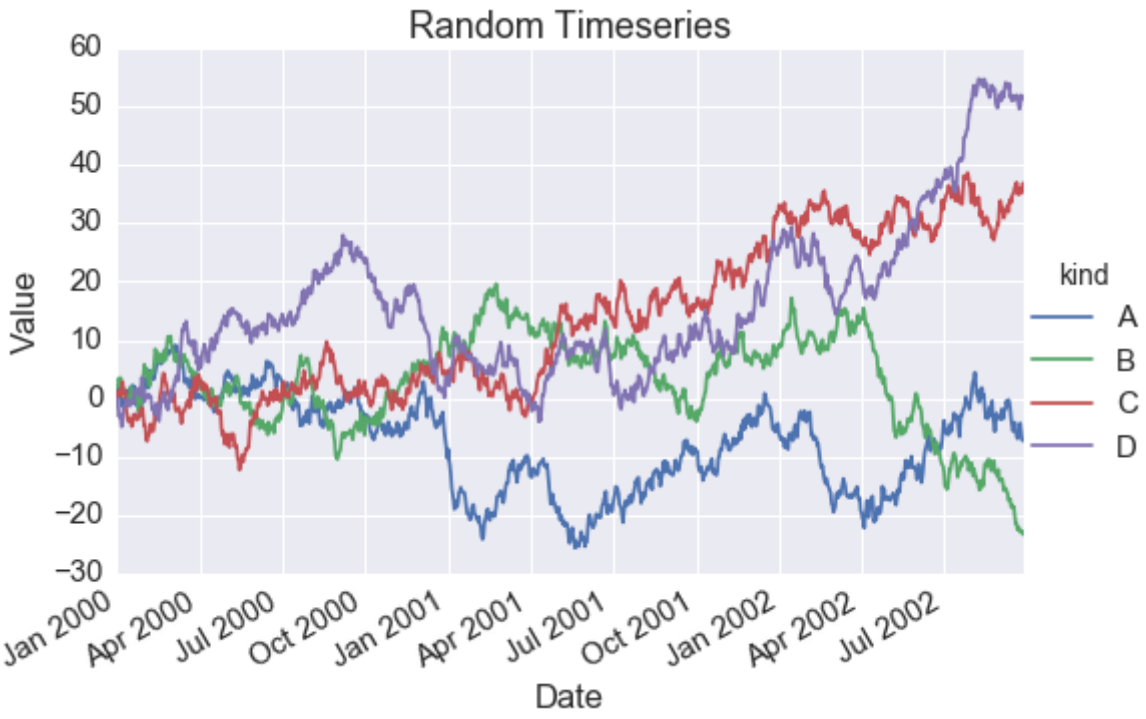
```
# PANDAS
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))
```

```
ax.legend(loc=2)
fig.autofmt_xdate()
```

pandas: 结果看上去完全一样，所以我就不展示了。

Seaborn（抽着烟，调整着贝雷帽）： 唔。看上去区区一个折线图就让你们做了这么多数据处理。我是说，for 循环和轴向旋转？这不是九十年代的微软 Excel（译者注：pivot table 即 Excel 的数据透视表）。我在国外学到一个叫做 FacetGrid 的东西。你们大概从来没有听说过……

```
# SEABORN
g = sns.FacetGrid(ts, hue='kind', size=5, aspect=1.5)
g.map(plt.plot, 'dt', 'value').add_legend()
g.ax.set(xlabel='Date',
         ylabel='Value',
         title='Random Timeseries')
g.fig.autofmt_xdate()
```



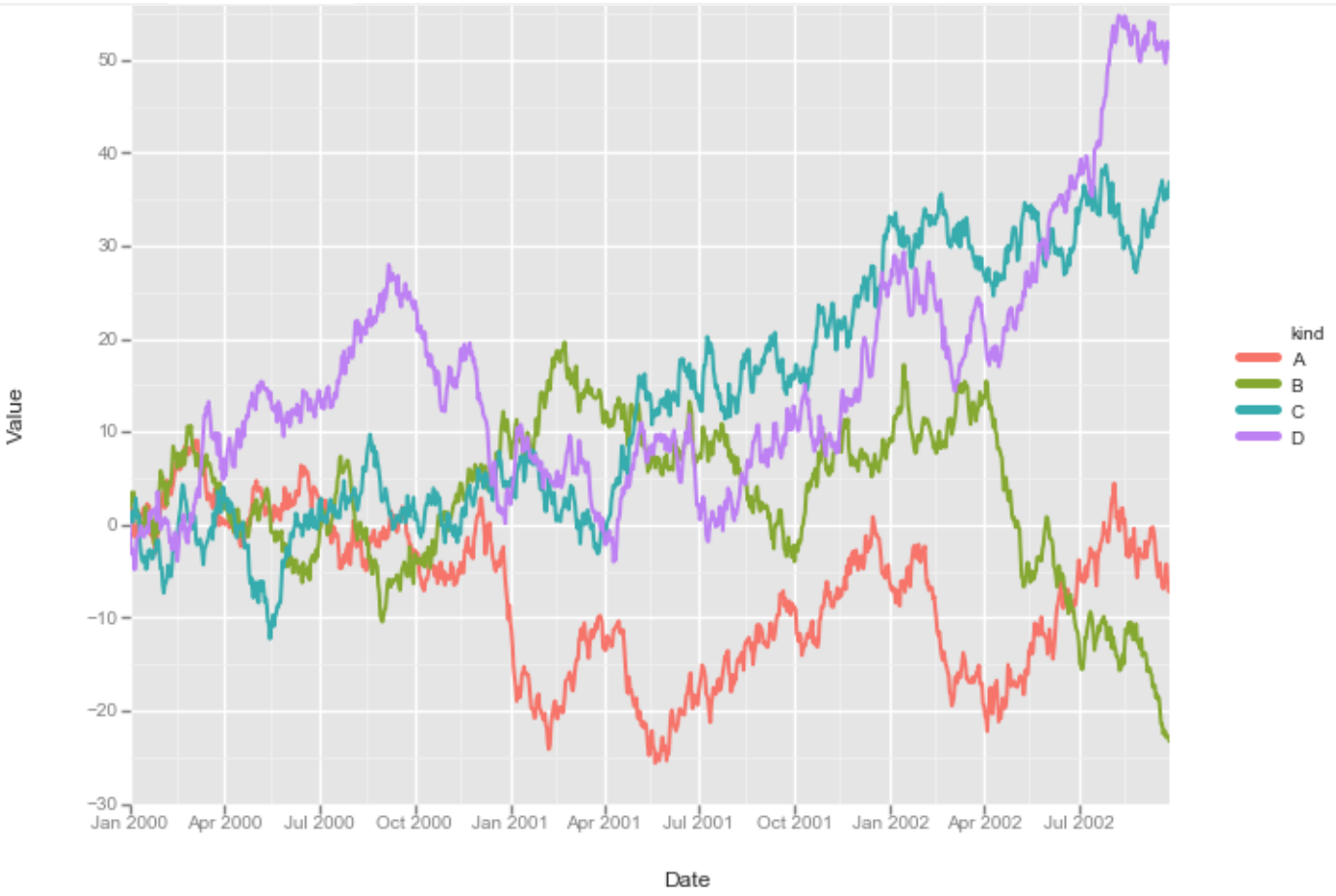
SB: 看懂了吗？直接给 FacetGrid 传入未处理的整洁数据。在这里，将 `kind` 赋给 `hue` 参数的意

ggplot: 哇，赞！我的方法和她差不多，但是我做起来更像我的大哥。你们听过他吗？他超级酷——

SB: 谁邀请了这个孩子？

GG: 快来看看！

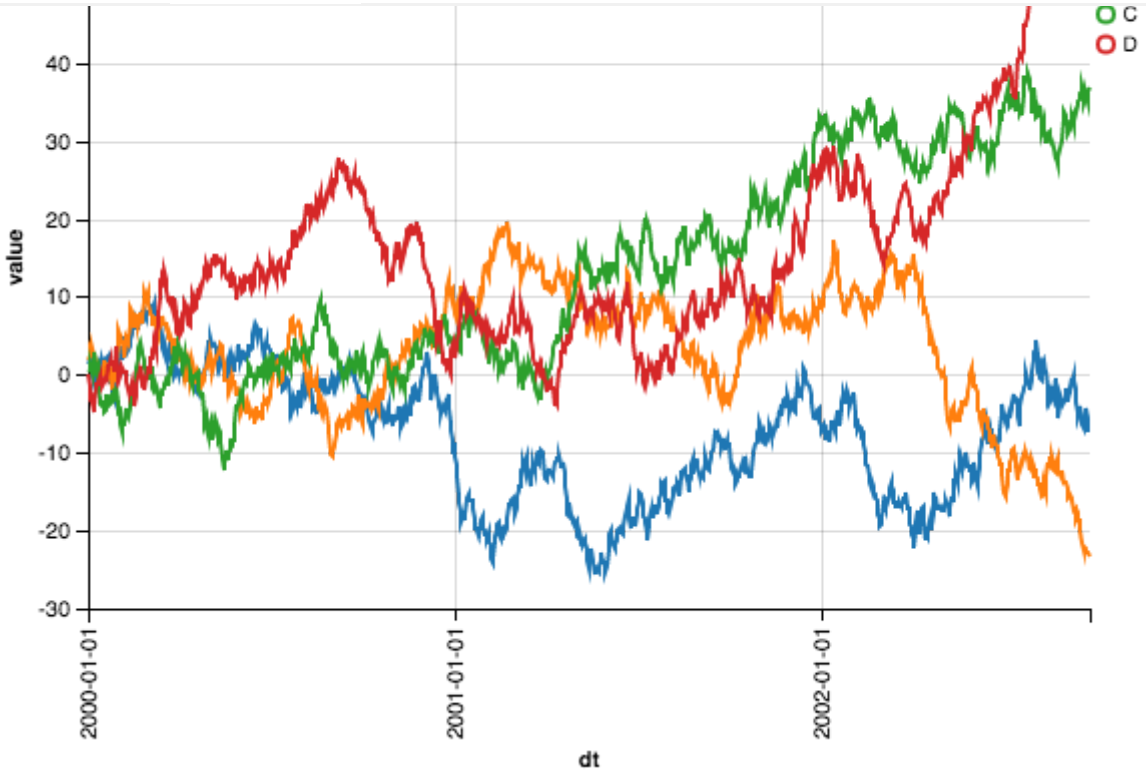
```
# GGPLOT
fig, ax = plt.subplots(1, 1, figsize=(7.5, 5))

g = ggplot(ts, aes(x='dt', y='value', color='kind')) + \
        geom_line(size=2.0) + \
        xlab('Date') + \
        ylab('Value') + \
        ggtitle('Random Timeseries')
g
```



GG (拿起 Hadley Wickham 写的 《ggplot2》读出声来): 每一幅图都由数据（比如 `ds` ），图形映射（比如 `x` ， `y` 和 `color` ）和几何图形（比如 `geom_line` ）组成，而后者将数据和图形映射转换成真正的可视化。

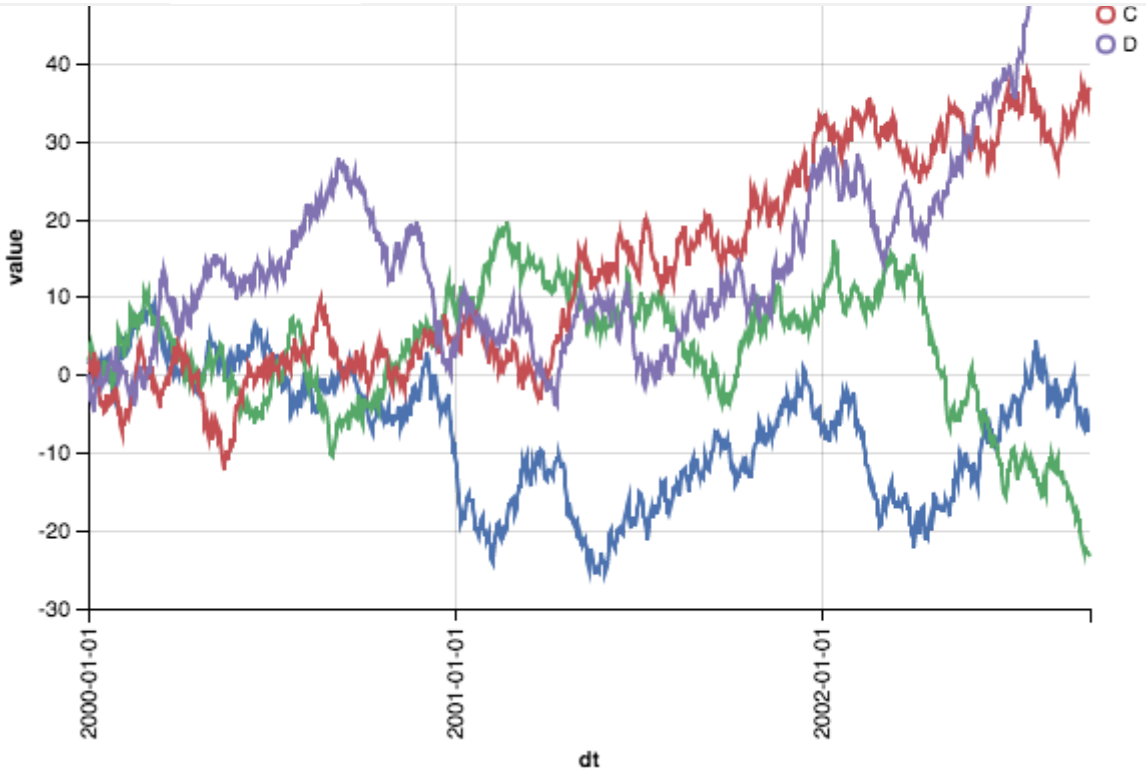
Altair: 没错，我也是这么做的。

```
# ALTAIR
c = Chart(ts).mark_line().encode(
    x='dt',
    y='value',
    color='kind'
)
c
```



ALT: 给我的 Chart 类同样的数据，告诉它你要哪种可视化：这里就是 `mark_line` 。然后指定想要的图形映射：x 轴是 `data` ，y 轴是 `value` ；因为我们想要按 `kind` 分组，所以把 `kind` 传给 `color` 。就跟你一样，GG（**拨乱 GG 的头发**）。哦，这样一来，要用你们都用的配色方案也轻而易举了：

```
# ALTAIR

# cp corresponds to Seaborn's standard color palette
c = Chart(ts).mark_line().encode(
    x='dt',
    y='value',
    color=Color('kind', scale=Scale(range=cp.as_hex()))
)
c
```



MPL 害怕又惊讶地盯着

第一场的分析

除了混蛋的 matplotlib [3]，还有一些要点值得注意。

- 用 matplotlib 和 pandas 的时候，要么得多次调用 `plot` 函数（比如每个 for 循环里面），要么得对数据进行处理才能更好适用于 plot 函数（比如轴向旋转）。（也就是说我们在第二场还会见到其他的技术。）
- （说实话，我从来不觉得这是个大问题，直到我遇到了 R 语言使用者。他们看到我都惊呆了。）
- 与之相反，ggplot 和 Altair 用的是类似声明式「图形语法」的方法去解决这种简单问题：给「主」函数（ggplot 中的 `ggplot` 和 Altair 中的 `Chart`）传入整洁的数据集。然后定义一组图形映射（x，y 和 color）来说明数据该如何映射到图形上（比如视觉标记做了很多努力以便更好地传达信息）。只要使用这些图形（ggplot 的 [illegible] 和 Altair 的 [illegible]）数据

映射并不是图形映射，只是函数映射：数据集中的每一个 `hue` 都会调用 matplotlib 的 plot 函数， `dt` 和 `value` 分别传给 x 和 y 参数。for 循环是不可见的底层实现。

- 也就是说，尽管图形映射需要两个独立的步骤，比起命令式的思维方式，我还是更喜欢图形映射（至少在画图时如此）。

数据说明

（在第二场到第四场，我们会处理著名的「鸢尾花」数据集（在代码中用 `df` 表示）。它包含了四个数字列，对应不同的测量，还有一个类别列，表明它是三种鸢尾花中的哪一种。下面是预览：

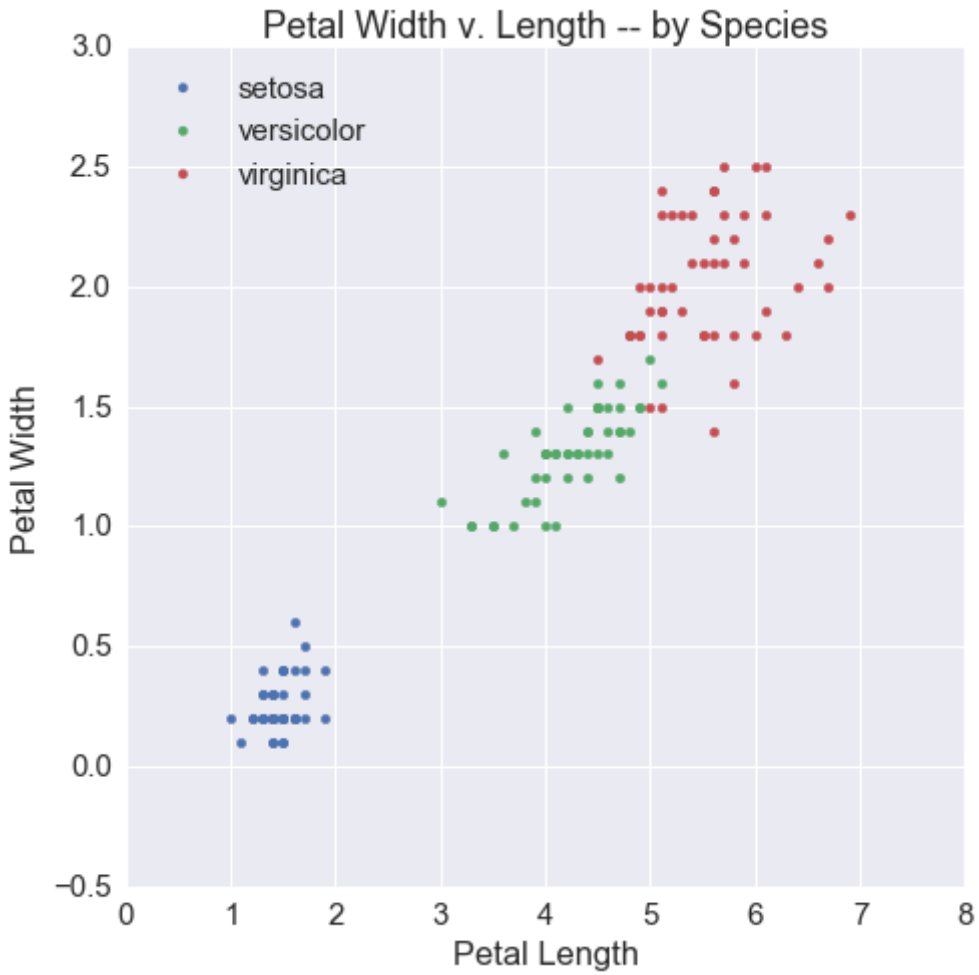
花瓣长度	花瓣宽度	萼片长度	萼片宽度	品种
0	1.4	0.2	5.1	3.5
1	1.4	0.2	4.9	3.0
2	1.3	0.2	4.7	3.2
3	1.5	0.2	4.6	3.1
4	1.4	0.2	5.0	3.6

第二场：如何画散点图？

MPL（看上去有点震惊）： 我是说，你可以继续用 for 循环，当然了。这样也没什么问题。当然。懂了吗？（**压低声音小声说**）只要记得显式地设定好颜色变量，不然所有的点都是蓝的……

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(7.5, 7.5))

for i, s in enumerate(df.species.unique()):
    tmp = df[df.species == s]
```

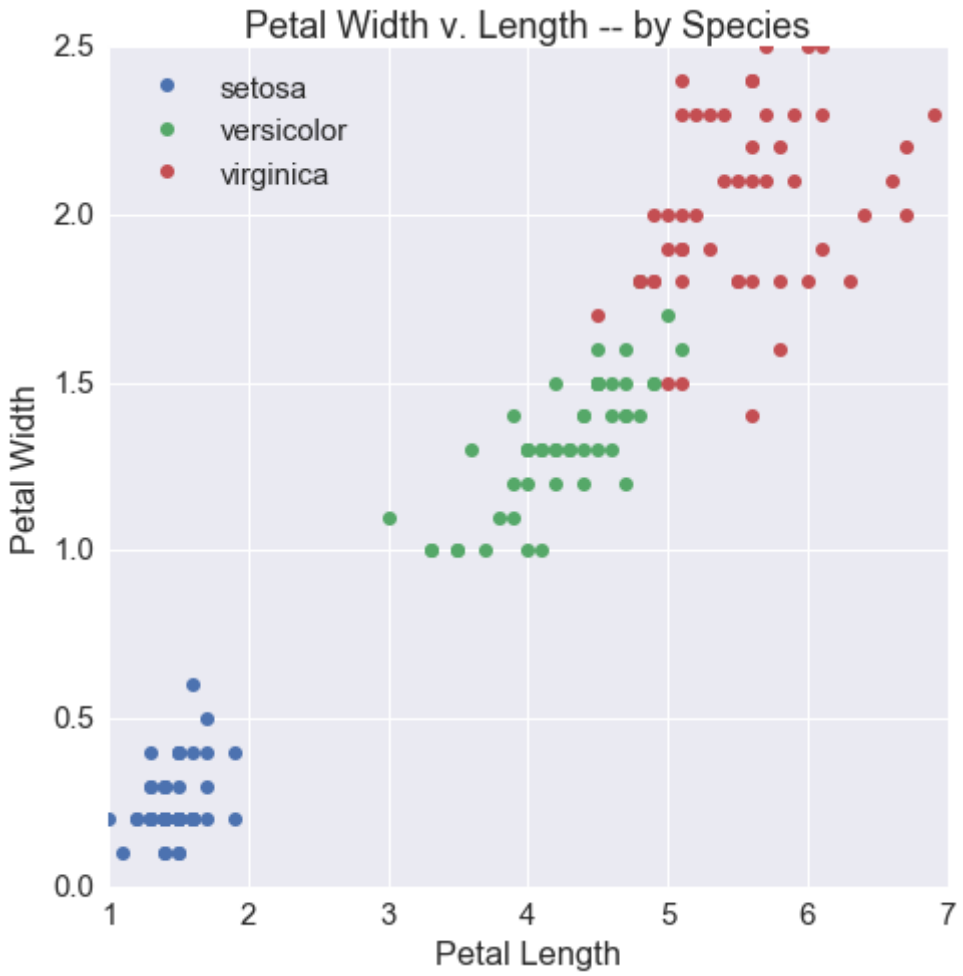
```
        ylabel='Petal Width',
        title='Petal Width v. Length -- by Species')

ax.legend(loc=2)
```



MPL: 可是，呃，（**假装充满自信**）我有个更好的主意！看这个：

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(7.5, 7.5))

def scatter(group):
    plt.plot(group['petalLength'],
             group['petalWidth'],
             'o', label=group.name)
```

```
        title='Petal Width v. Length -- by Species')

ax.legend(loc=2)
```



MPL: 我在这定义了 `scatter` 函数。它用 pandas 的 groupby 对象得到分组，然后在 x 轴上画出花瓣长度，y 轴则是花瓣宽度。每组都如此处理一番！厉害吧！

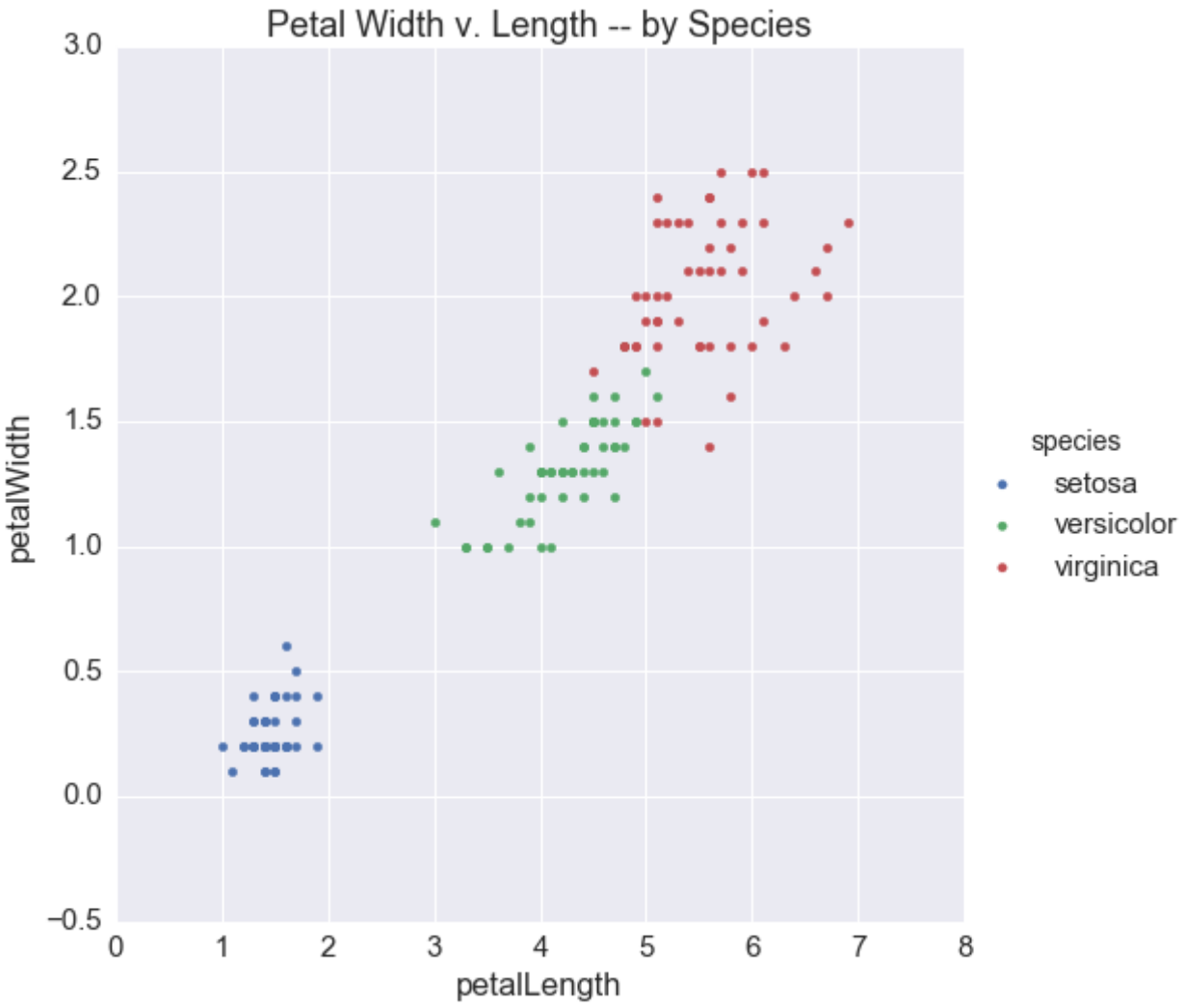
P: 真不错，Mat！真不错！基本上和我的方法差不多，所以我就坐这里不展示了。

SB (咧嘴笑): 这次怎么没用轴向旋转？

P: 嗯，这个例子里要用轴向旋转的话比较复杂。因为不像处理时序数据一样有一个通用的索引，所以……

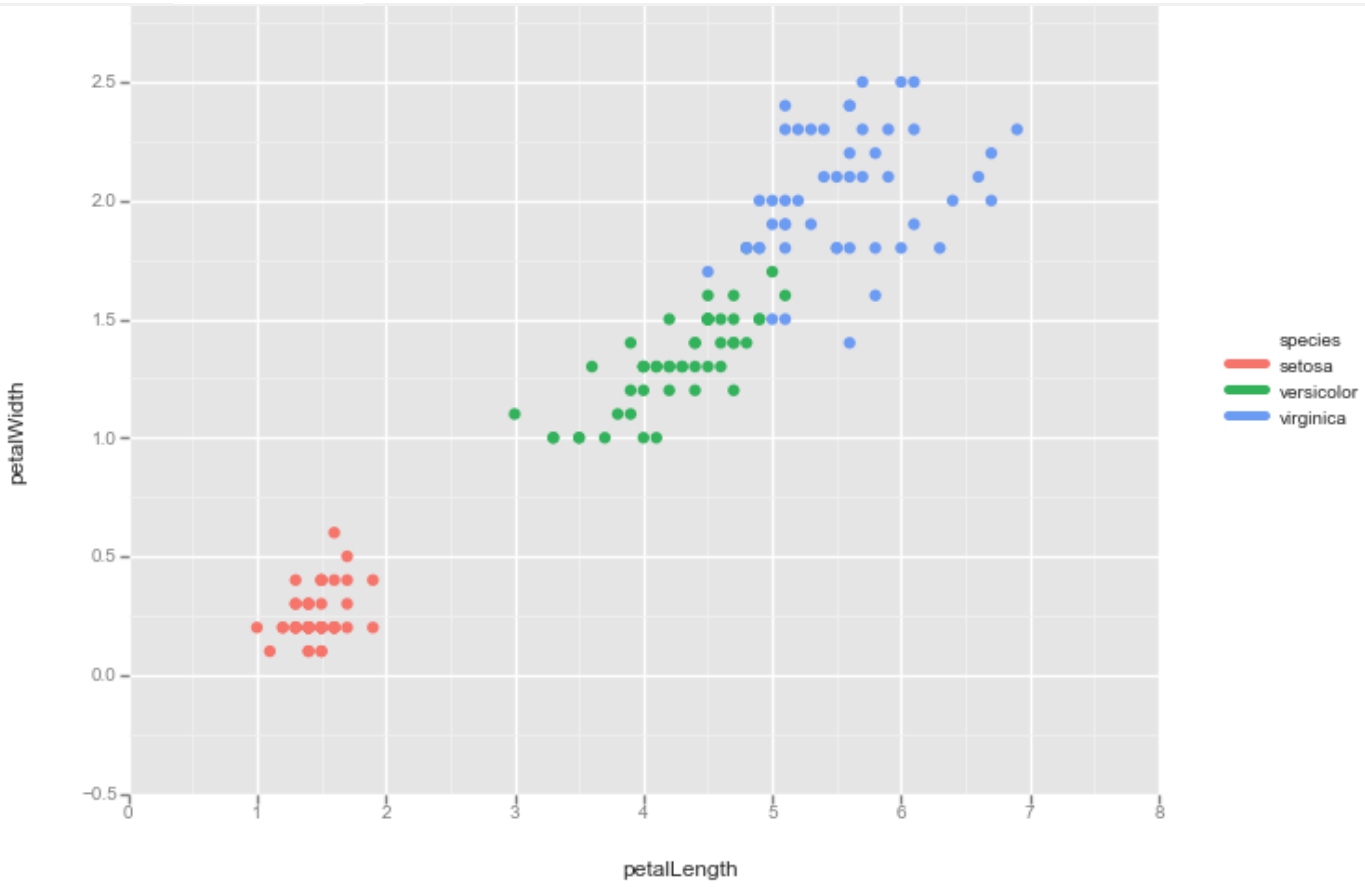
MPL: 喊！我们没必要跟她解释

```
# SEABORN
g = sns.FacetGrid(df, hue='species', size=7.5)
g.map(plt.scatter, 'petalLength', 'petalWidth').add_legend()
g.ax.set_title('Petal Width v. Length -- by Species')
```



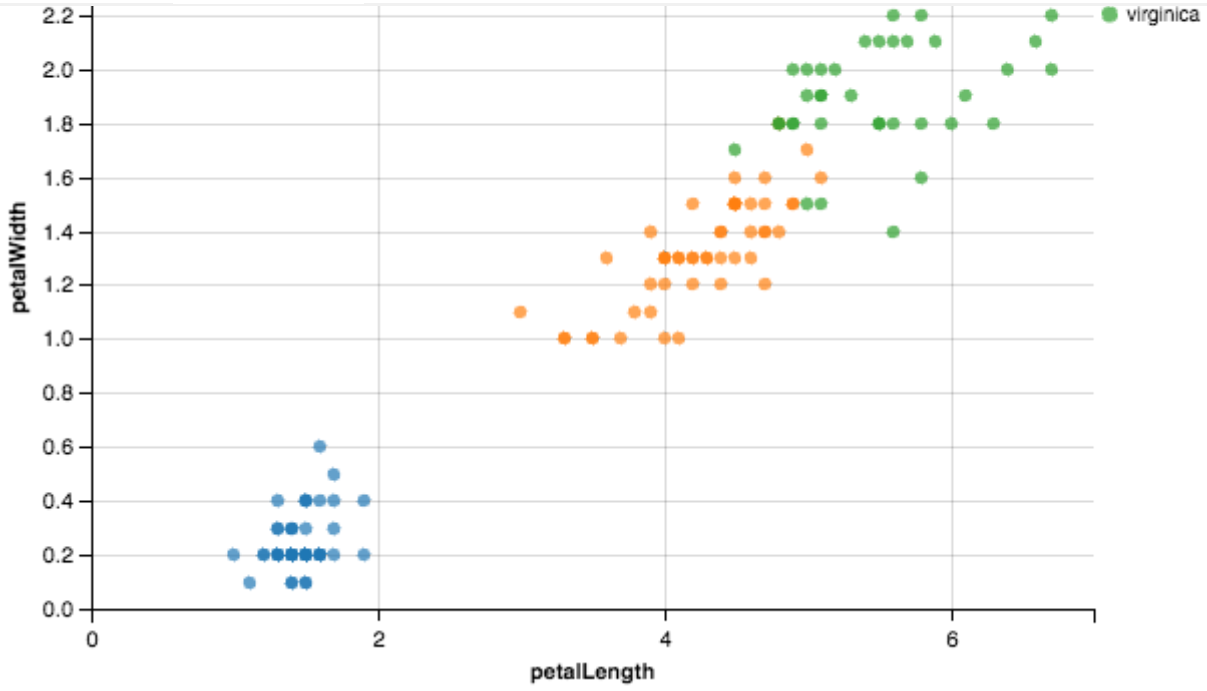
GG: 对！对！就是这样！我的写法就是把 `geom_line` 换成 `geom_point` ！

```
# GGPLOT
g = ggplot(df, aes(x='petalLength',
                   y='petalWidth',
                   color='species')) + \
        geom_point(size=40.0) + \
```



ALT (一脸茫然): 是的，只要把 `mark_line` 换成 `mark_point` 。

```
# ALTAIR
c = Chart(df).mark_point(filled=True).encode(
    x='petalLength',
    y='petalWidth',
    color='species'
)
c
```



第二场的分析

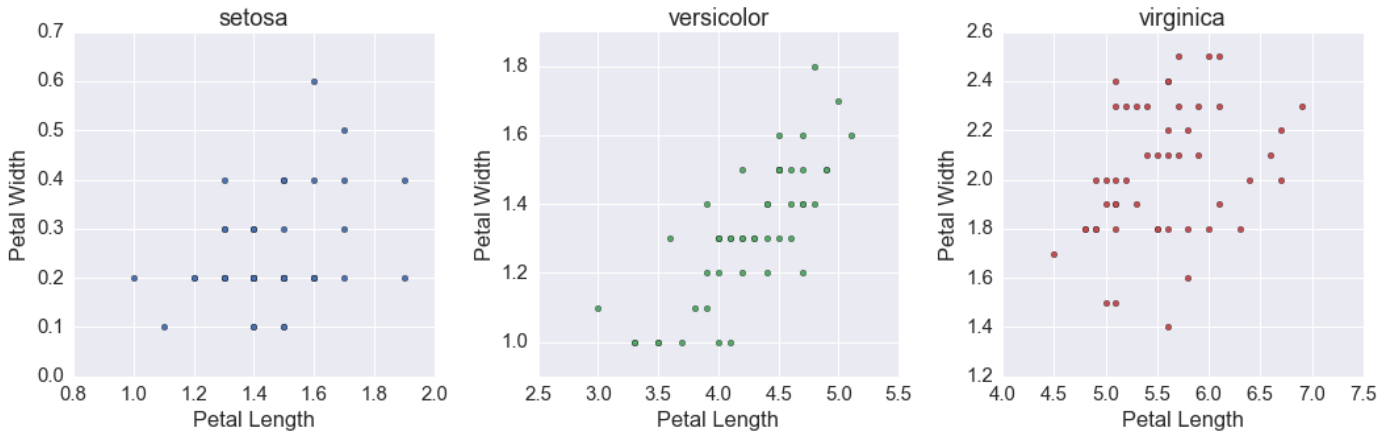
- 到这儿，用数据构建 API 的潜在难题变得清晰了。尽管 pandas 的轴向旋转处理时序数据时非常方便，处理这个例子却力不从心了。
- 公平地说， `group by` 方法是可以推导出来的，而 for 循环就更容易推出来了；但是这样一来就要有更多自定义的逻辑，也就意味着更多的工作：Seaborn 已经好心帮你做好了，不然你还得自己造轮子。
- 反过来说，Seaborn，ggplot 和 Altair 都明白散点图在很多方面就是没有假设的折线图（尽管这些假设可能是无害的）。因此，第一场中的代码大都可以重用，但是得用新的几何对象（ggplot 和 Altair 分别用的是 `geom_point` 和 `mark_point` ）或者新的方法（比如 Seaborn 的 `plt.scatter` ）。在这个节点上，没有哪个库比其他库更方便，尽管我爱 Altair 优雅的简洁。

第三场：如何画分面的散点图？

取数据子集的方法来取相关的 Axes 对象的子集。

（重拾自信）我敢打赌你们各位没有更简单的方法！（举起双臂，差点打到了 pandas）

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

for i, s in enumerate(df.species.unique()):
    tmp = df[df.species == s]

    ax[i].scatter(tmp.petalLength, tmp.petalWidth, c=cp[i])

    ax[i].set(xlabel='Petal Length',
              ylabel='Petal Width',
              title=s)

fig.tight_layout()
```



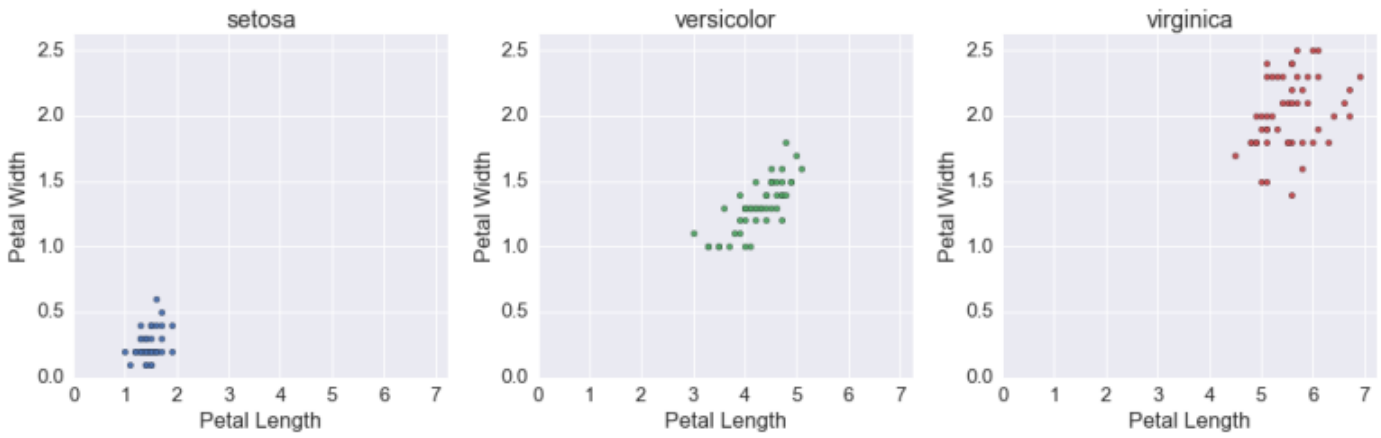
SB 和笑起来的 ALT 交换了目光；GG 仿佛听到笑话了笑了起来

MPL: 怎么啦？！

Altair: 老兄，看看你的 x 轴和 y 轴。所有图像的坐标轴范围都不一样。

MPL (脸红了): 呃，是，当然啊。我就是想看看你们有没有注意听我说话。你当然可以在 `subplot` 函数中指定坐标轴范围，保证所有的子图坐标轴范围是统一的。

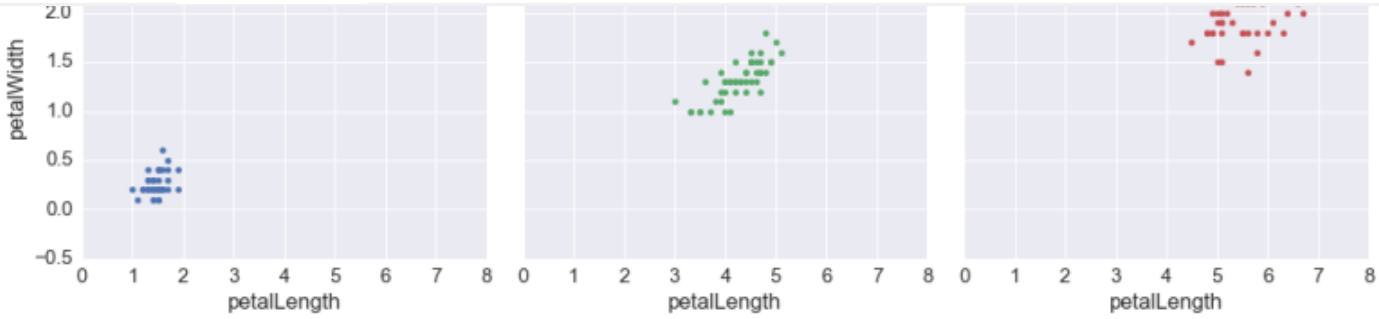
```
    tmp = df[df.species == s]

    ax[i].scatter(tmp.petalLength,
                  tmp.petalWidth,
                  c=cp[i])

    ax[i].set(xlabel='Petal Length',
              ylabel='Petal Width',
              title=s)

fig.tight_layout()
```



P（叹气）： 我也是这么做的。跳过我吧。

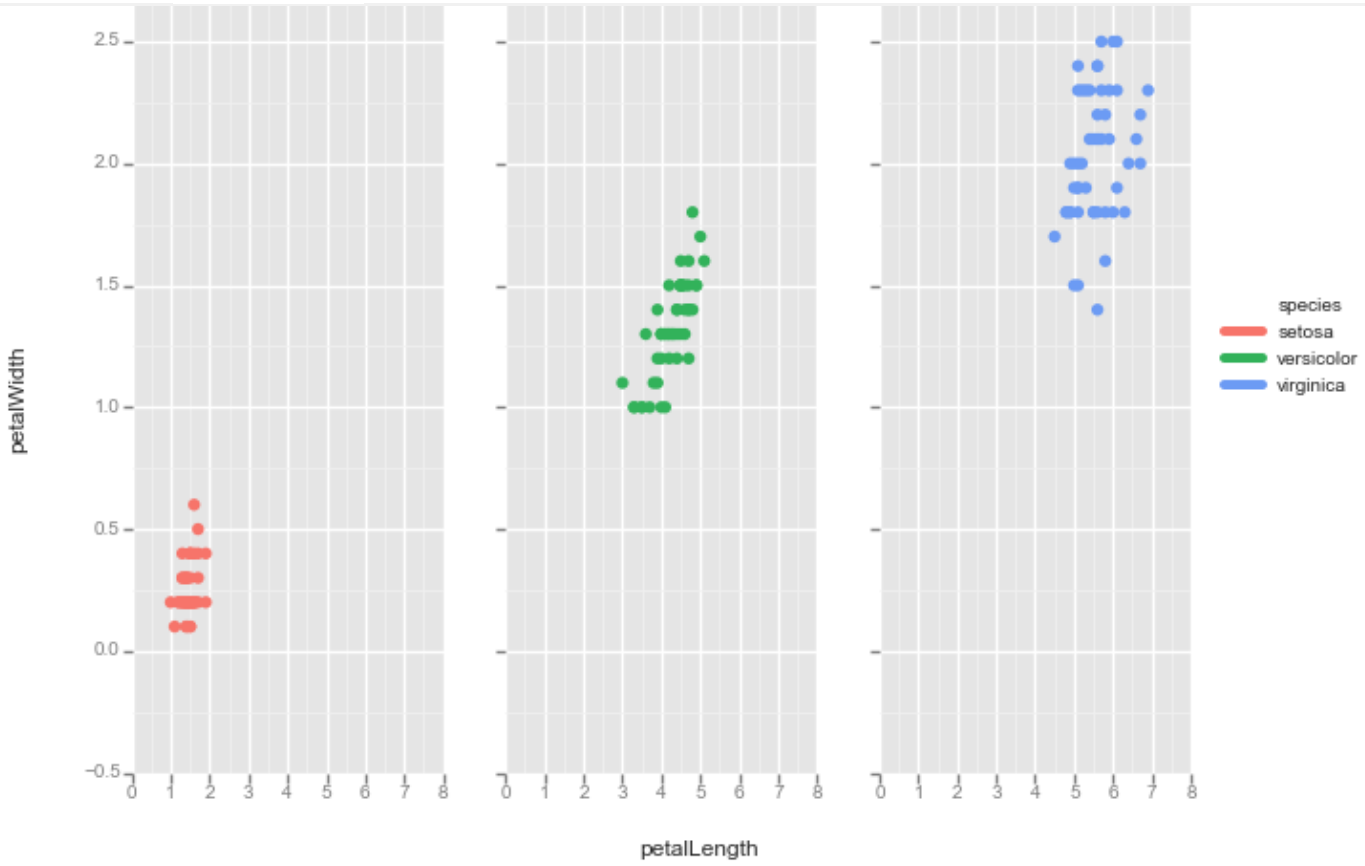
SB: 改写 FacetGrid 然后用在这个例子上很简单。就像使用 `hue` 变量一样，我们可以简单加一个 `col` 变量（比如 colum）。这会告诉 FacetGrid 不仅给每个种类一个唯一的颜色，还把每个种类都画在唯一的子图上，按列排列。（只要将 `col` 变量换成 `row` 就可以按行排列。）

```
# SEABORN
g = sns.FacetGrid(df, col='species', hue='species', size=5)
g.map(plt.scatter, 'petalLength', 'petalWidth')
```



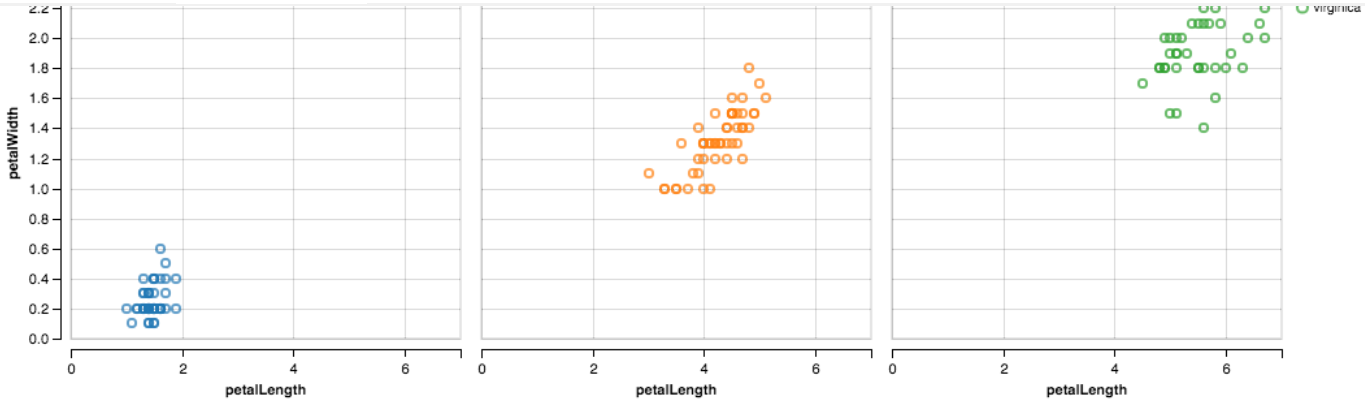
GG: 哦，这和我的做法不同（**再一次拿起《ggplot2》开始读**）。看，分面和图形映射本质上是两个不同的步骤，我们不应该一时疏忽把它们混为一谈。因此，我们接着用之前的代码这次加上 `facet_grid` 层，也就是显式地用类别进行分面。（**开心地合上书**）至少我大哥是这么说的！你们听到他了吗？在书里。他真酷啊[4]。

```
# GGPLOT
g = ggplot(df, aes(x='petalLength',
                   y='petalWidth',
                   color='species')) + \
        facet_grid(y='species') + \
        geom_point(size=40.0)
g
```



ALT: 我这里采用更具 Seaborn 风格的方法。具体地说，我给编码函数加了一个 `column` 参数。也就是说我也做了一些新工作：第一，虽然 `column` 参数可以接受一个简单的字符串变量，实际上我传给它的是 Column 对象，如此我可以自定义标题了。第二，我用了自定义的 `configure_cell` 方法，如果不用的话子图会变得特别巨大。

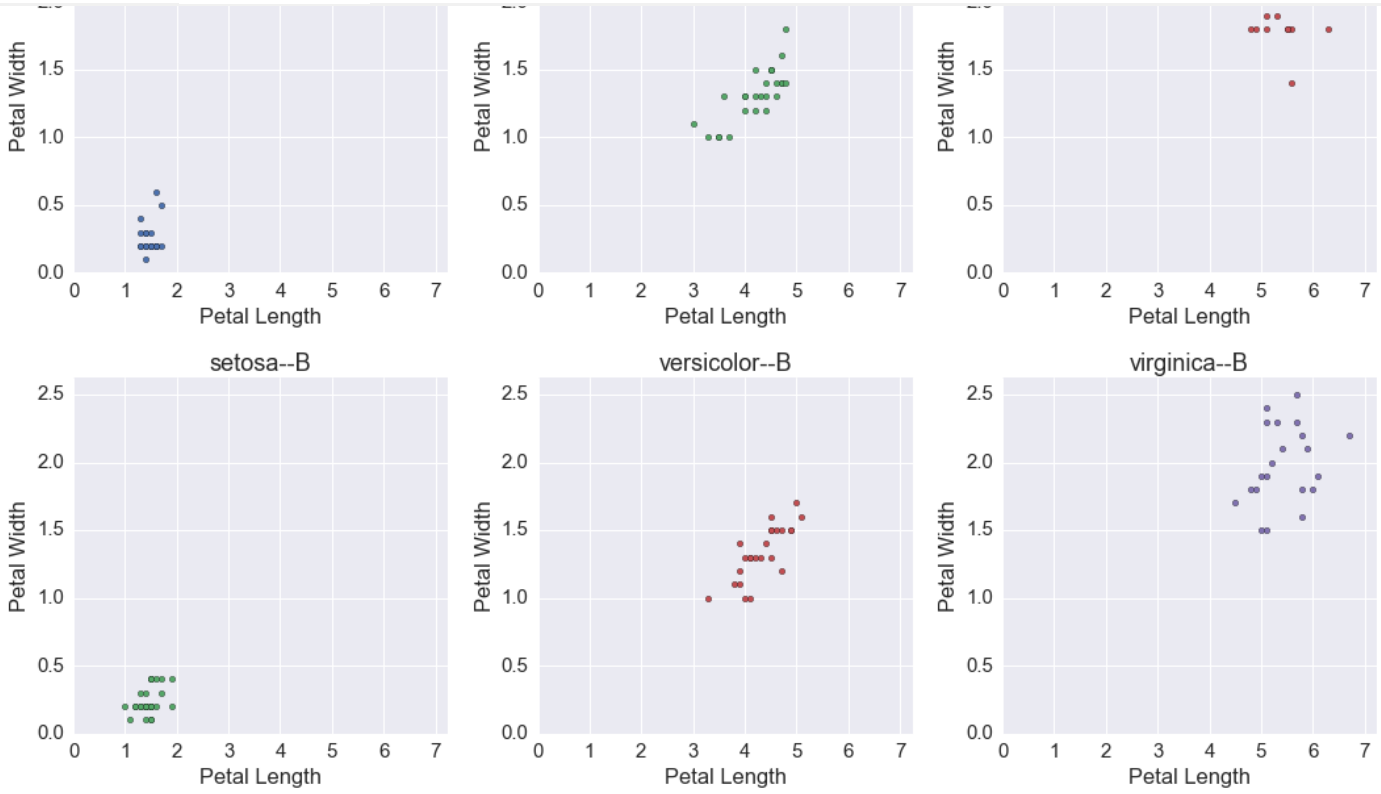
```
# ALTAIR
c = Chart(df).mark_point().encode(
    x='petalLength',
    y='petalWidth',
    color='species',
    column=Column('species',
                  title='Petal Width v. Length by Species')
)
c.configure_cell(height=300, width=300)
```



第三场的分析

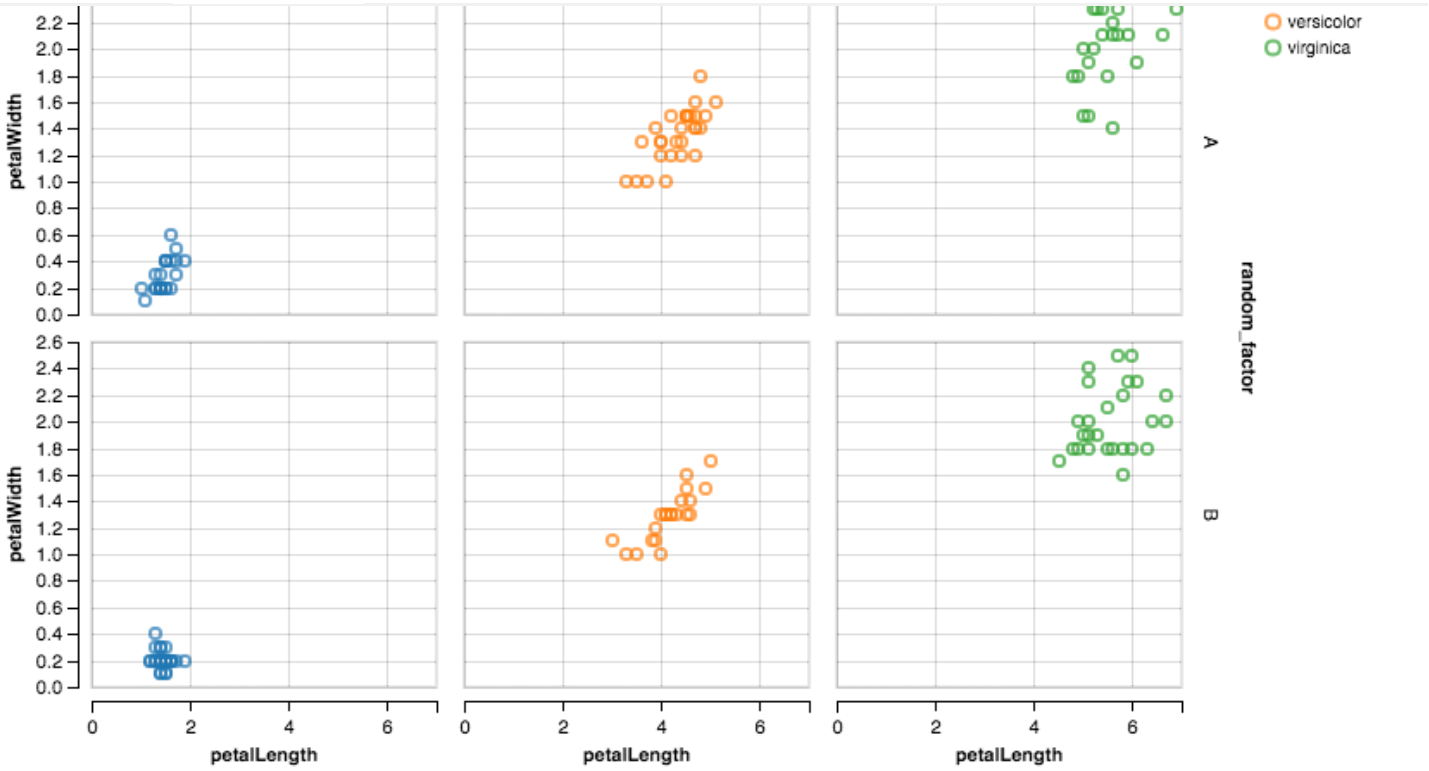
- matplotlib 说得很清楚：这个例子中，他的代码根据分类对数据进行分面的思路和上面的其他方案是一样的；假如你的脑袋可以搞清楚那些 for 循环的话，你可以再试试下面这段代码。但是我可没有让他再搞出更复杂的东西出来，比如 2 x 3 的网格。不然他就得像下面这样干：

```
# MATPLOTLIB
fig, ax = plt.subplots(2, 3, figsize=(15, 10), sharex=True, sharey=True)

# this is preposterous -- don't do this
for i, s in enumerate(df.species.unique()):
    for j, r in enumerate(df.random_factor.sort_values().unique()):
        tmp = df[(df.species == s) & (df.random_factor == r)]

        ax[j][i].scatter(tmp.petalLength,
                         tmp.petalWidth,
                         c=cp[i+j])

        ax[j][i].set(xlabel='Petal Length',
                     ylabel='Petal Width',
                     title=s + '--' + r)

fig.tight_layout()
```



- 为了用正规的可视化表达式：**呸**。如果用 Altair 的话一切都变得非常简单。

```
# ALTAIR
c = Chart(df).mark_point().encode(
    x='petalLength',
    y='petalWidth',
    color='species',
    column=Column('species',
                  title='Petal Width v. Length by Species'),
    row='random_factor'
)
c.configure_cell(height=200, width=200)
```



- 只比我们刚才用过的 `encode` 函数多一个变量！
- 幸运的是，把分面构建到可视化库框架中的好处是显而易见的。

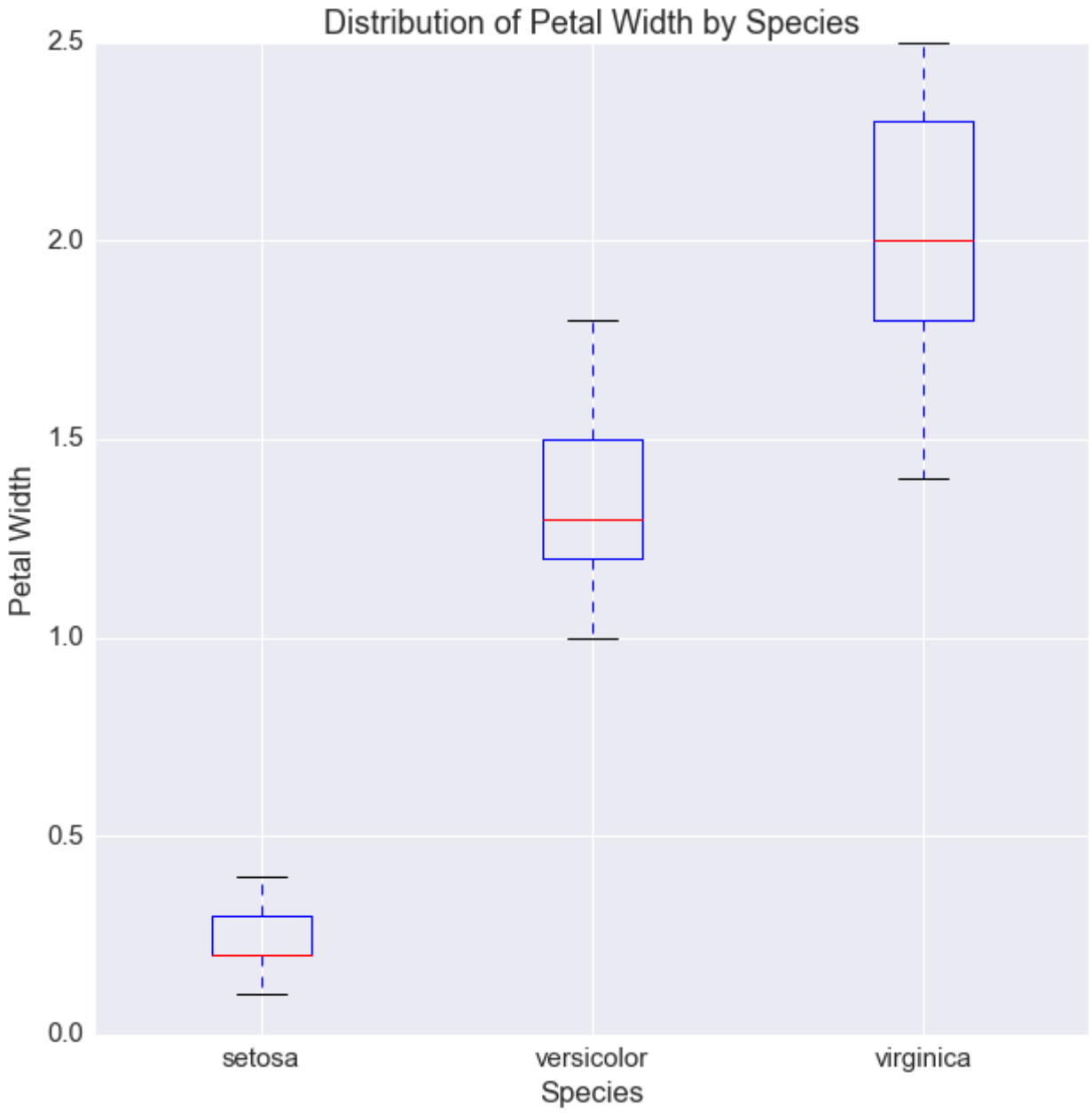
第二幕：分布和条形图

第四场：怎么可视化分布？

MPL（信心明显不足了）： 好吧，如果我们要画箱线图——我们真的要箱线图吗？——我知道怎么画。不过非常愚蠢；你肯定不会喜欢。不过我给 `boxplot` 方法传入一个数组组成的数组，每个数组就都会得到一个箱线图。你可能需要手动标注 X 轴的刻度。

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
```

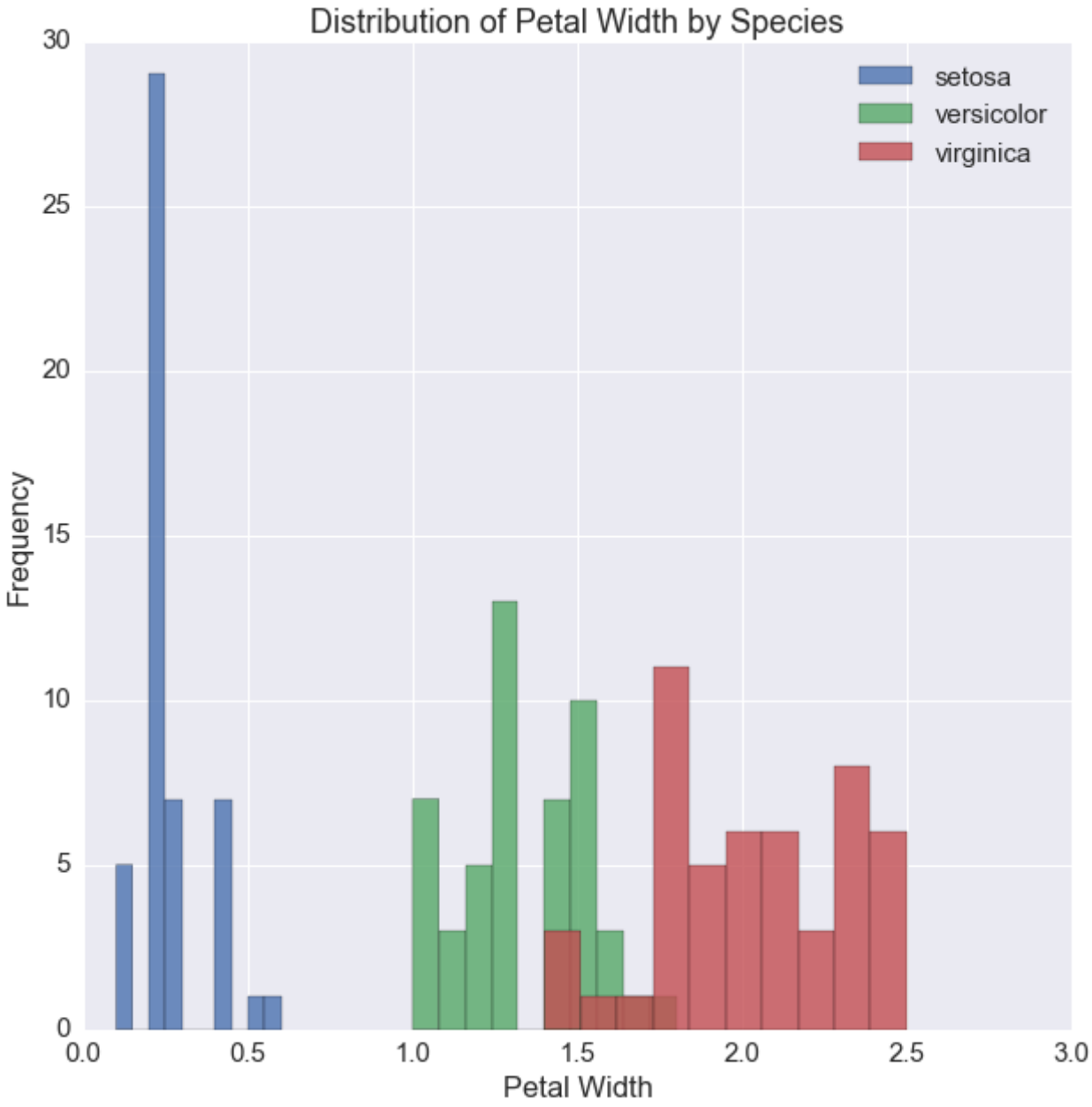
```
        ylabel='Petal Width',
        title='Distribution of Petal Width by Species')
```



MPL: 如果要画柱状图 —— 我们真的要画柱状图吗？ —— 我也有个方法可以用，你可以用之前提到的 for 循环或者 `group by` 。

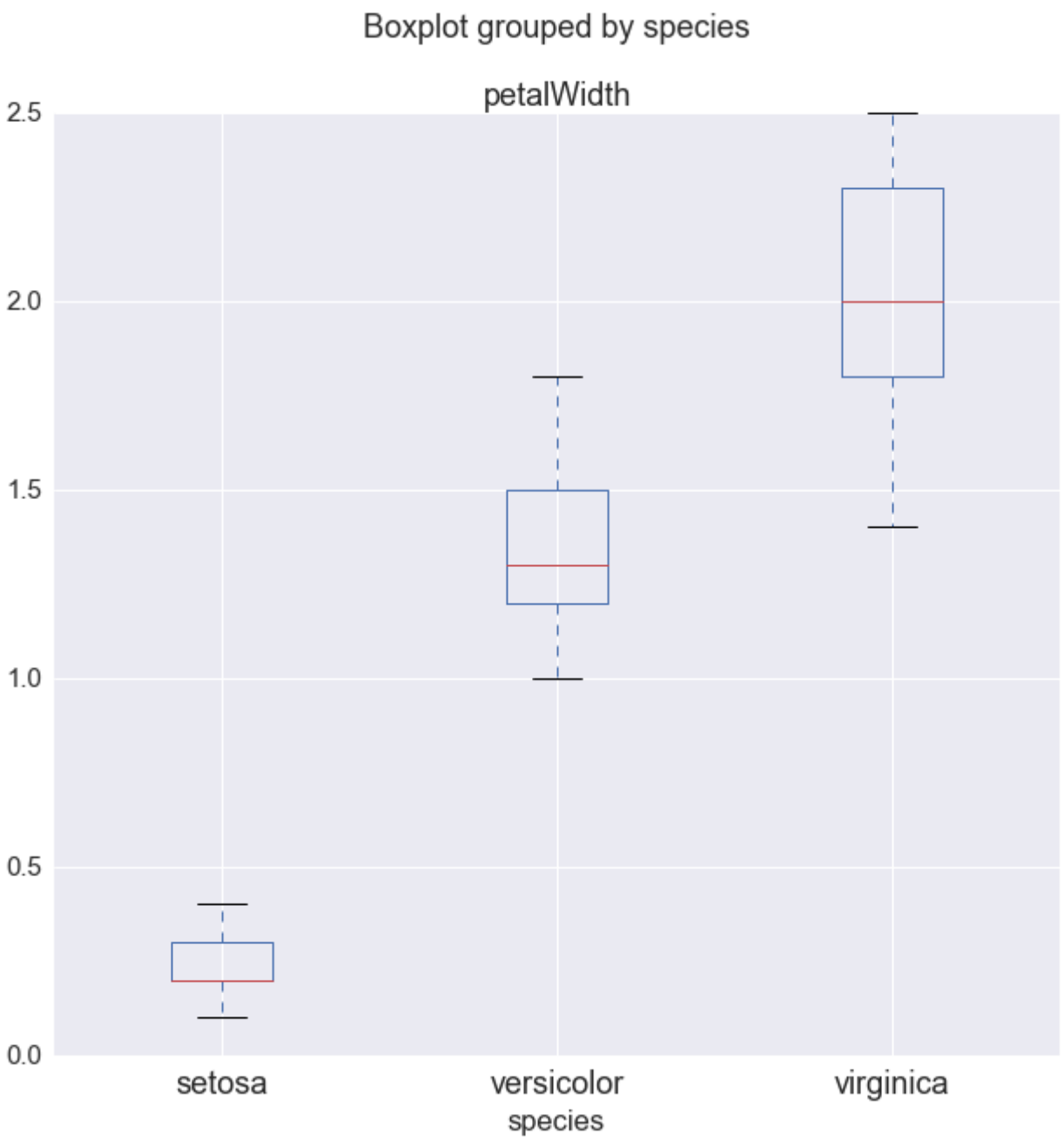
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
```

```
ax.set(xlabel='Petal Width',
       ylabel='Frequency',
       title='Distribution of Petal Width by Species')

ax.legend(loc=1)
```

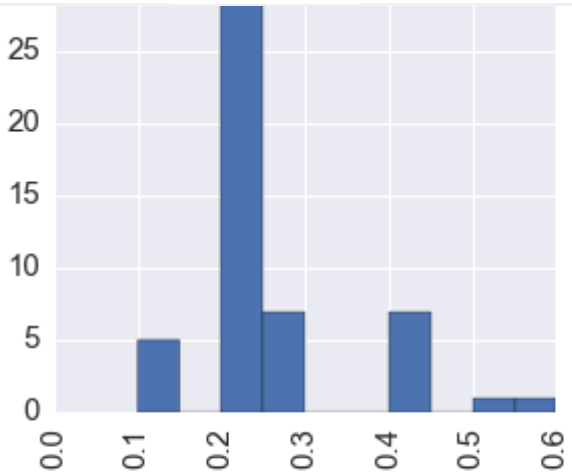


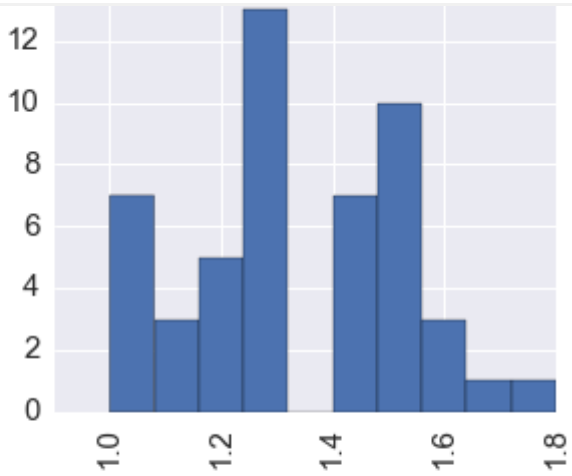
P (看上去不同寻常的骄傲）： 哈！哈哈哈哈！该我大显身手了！你们都觉得我一无是处，只是 `matplotlib` 的替罪羊。虽然我目前都只是套用他的 `plot` 方法，但我也拥有一些特殊的函数可以

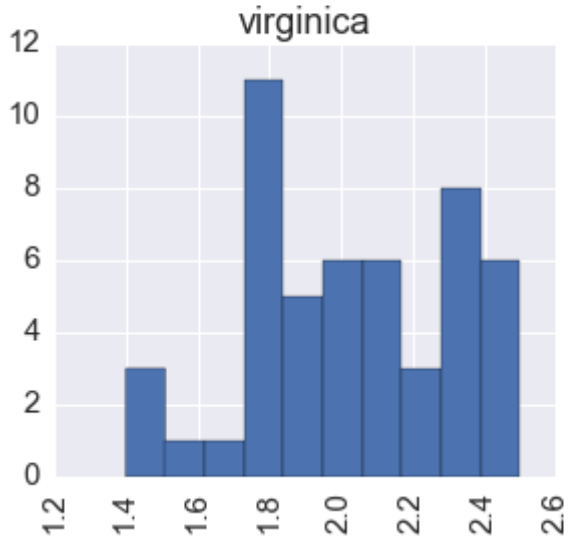
```
# PANDAS
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

df.boxplot(column='petalWidth', by='species', ax=ax)
```




```
# PANDAS
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
```



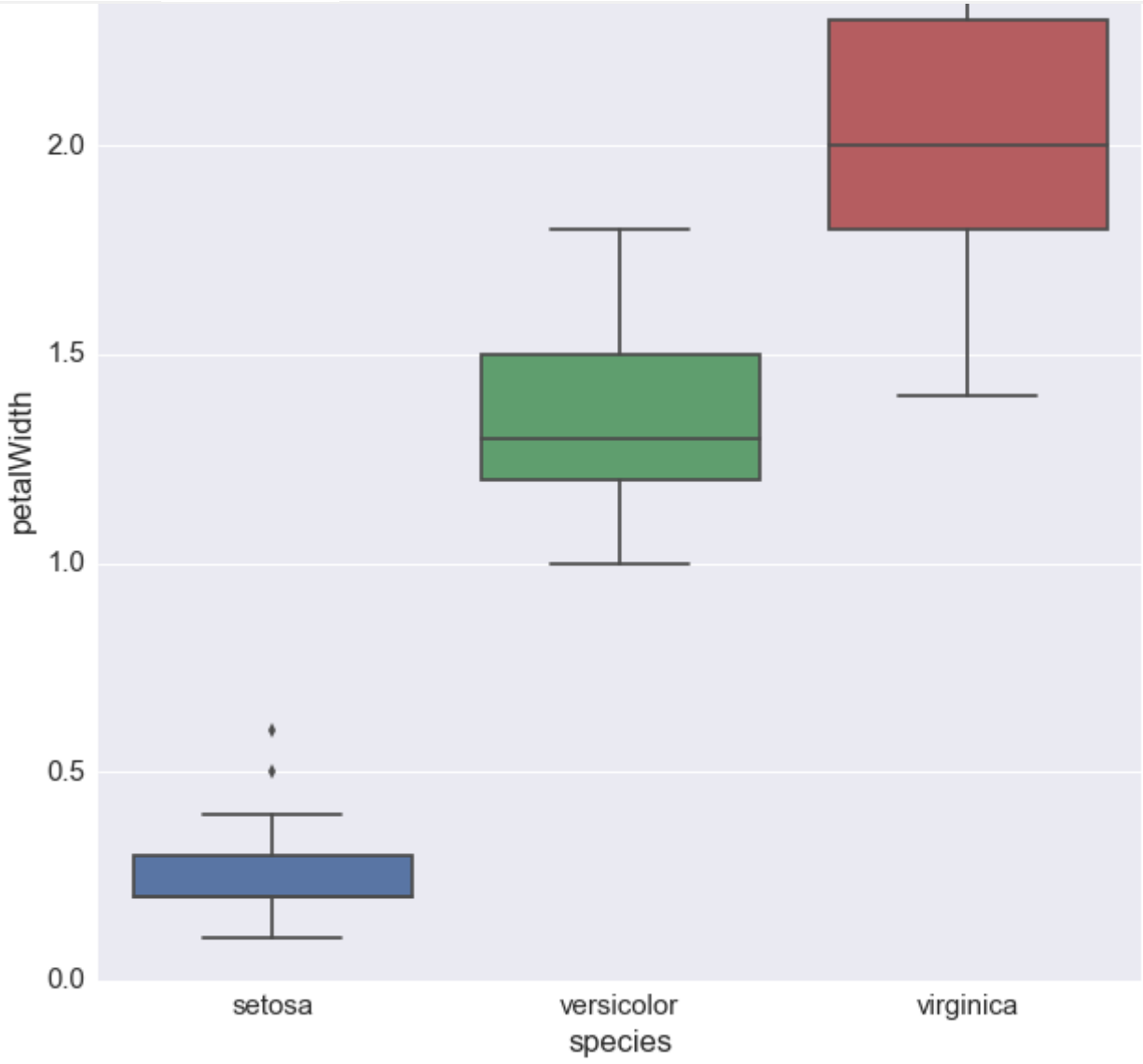




GG和ALT举手击掌然后祝贺P；高呼「棒极了！」，「就该这样！」，「就这么干！」

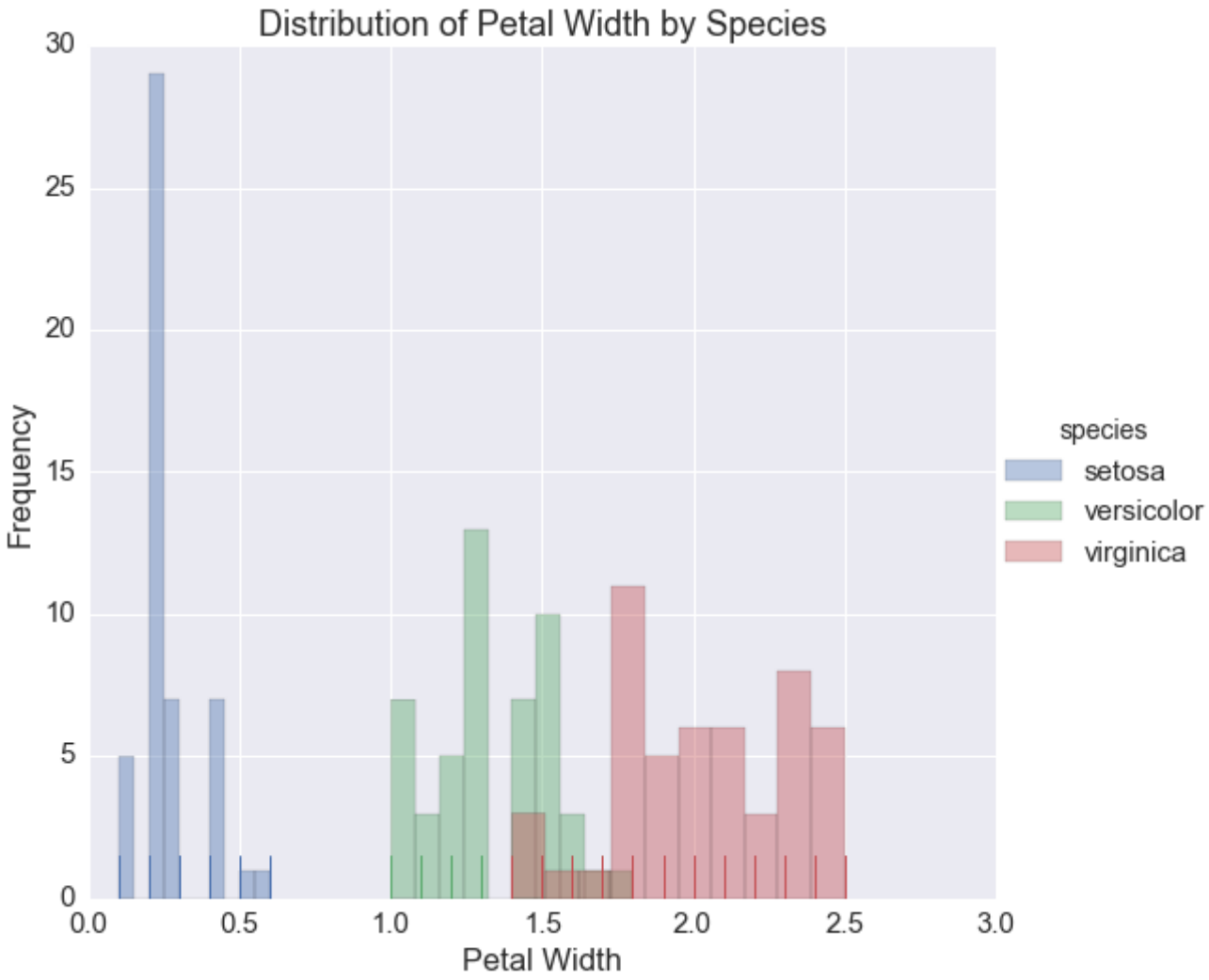
SB (假装很热情)： 喔喔喔。很赞。同时呢，分布对我非常重要，所以我为它准备了一些特殊方法。比如，我的 `boxplot` 方法只需要 x 变量、y 变量和数据就可以得到这个：

```
# SEABORN
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

g = sns.boxplot('species', 'petalWidth', data=df, ax=ax)
g.set(title='Distribution of Petal Width by Species')
```



SB: 这个图不错吧，我是说有人这么说过…… 不管了。我还有个特殊的分布方法叫 `distplot` 远不止条形图那么简单（傲慢的看了眼 pandas）。你可以用来画条形图，KDEs 和轴须图（rugplots）—— 甚至画在一起。比如把 `displot` 和 FacedGrid 结合起来，我就可以为每一种鸢尾花都画出直方轴须图：

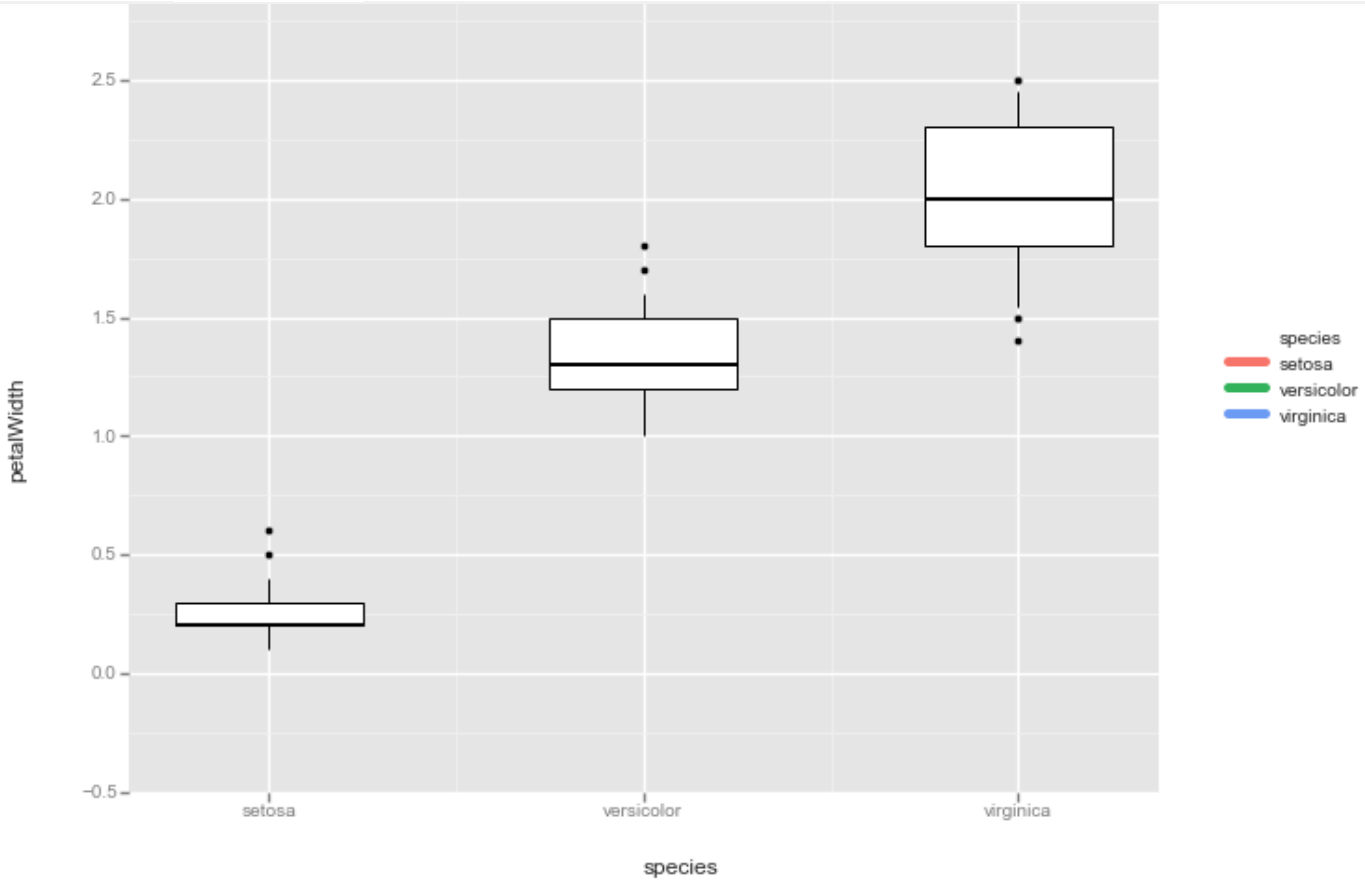
```
# SEABORN
g = sns.FacetGrid(df, hue='species', size=7.5)

g.map(sns.distplot, 'petalWidth', bins=10,
      kde=False, rug=True).add_legend()
```



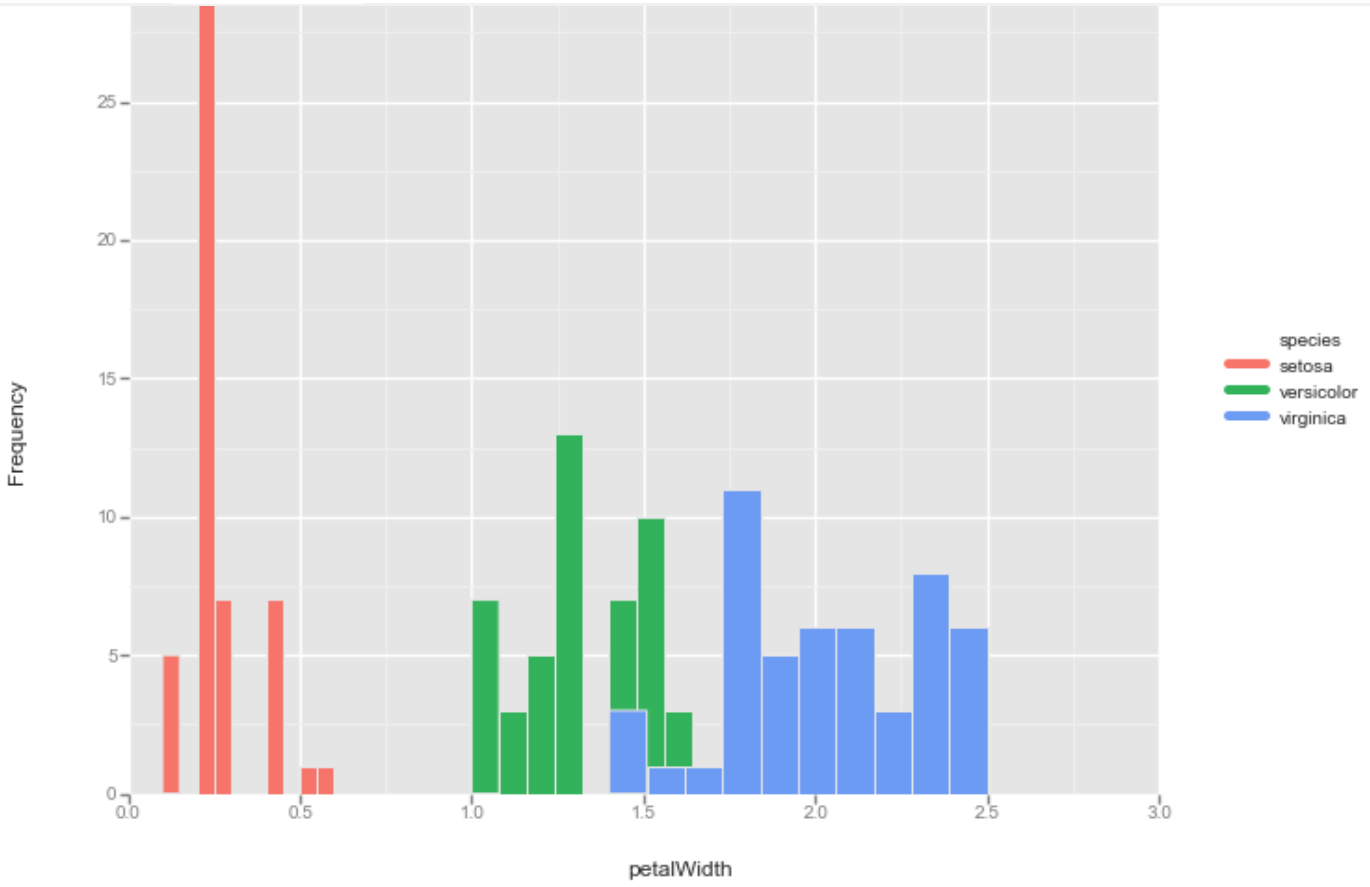
SB: 不过…… 管他呢。

GG: 这些只不过是新的几何对象！ `GEOM_BOXPLOT` 来画箱线图， `GEOM_HISTOGRAM` 来画直方图！换用它俩就行了！（**绕着餐桌跑了起来**）

```
# GGPLOT
g = ggplot(df, aes(x='species',
                   y='petalWidth',
                   fill='species')) + \
        geom_boxplot() + \
        ggtitle('Distribution of Petal Width by Species')
g
```




```
# GGPLOT
g = ggplot(df, aes(x='petalWidth',
                   fill='species')) + \
        geom_histogram() + \
        ylab('Frequency') + \
        ggtitle('Distribution of Petal Width by Species')
g
```

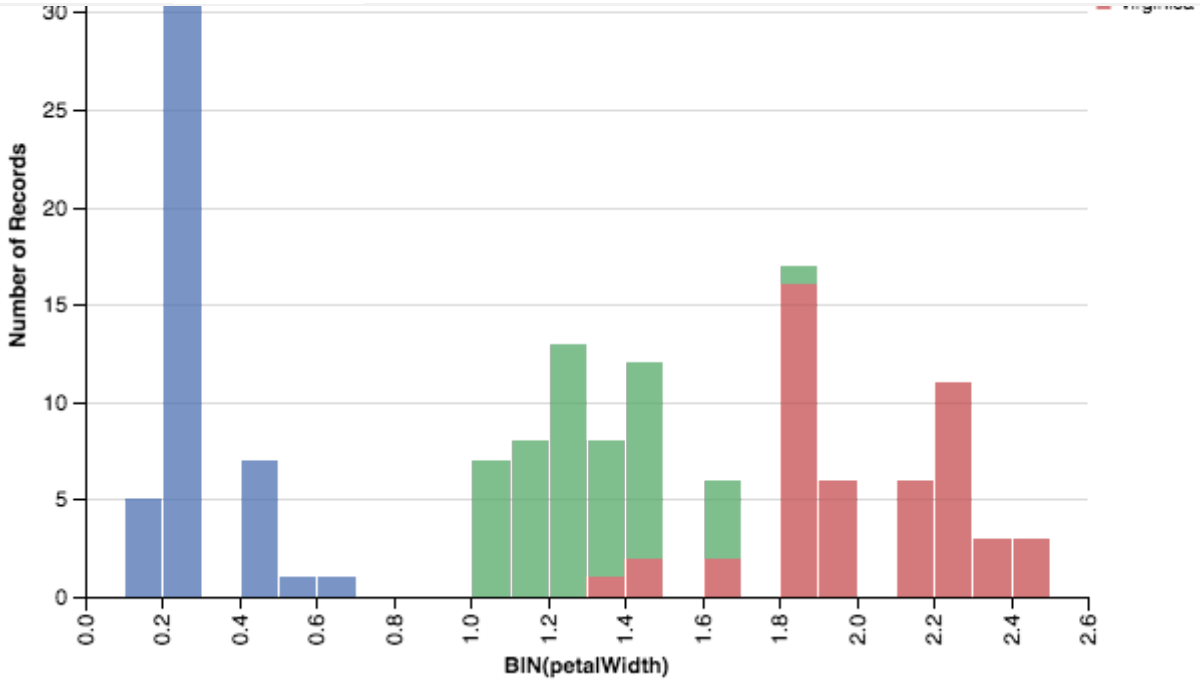


ALT（看上去坚定又自信）： 我要忏悔……

四周安静了下来 —— GG停了下来，把盘子撞到了地上。

ALT：（沉重地喘气） 我……我……我不会画箱线图。从来没学过怎么画，不过我相信我的源语言 JavaScript 的语法不支持箱线图肯定是有原因的。不过我会画直方图……

```
# ALTAIR
c = Chart(df).mark_bar(opacity=.75).encode(
    x=X('petalWidth', bin=Bin(maxbins=30)),
    y='count(*)',
    color=Color('species', scale=Scale(range=cp.as_hex()))
)
c
```



ALT: 乍一看代码会觉得有点怪，但是不要担心。这里实际是在说：「嘿，直方图事实上就是条形图」。X 轴对应着 `bin` ，我们可以用 `Bin` 类来定义；同时 y 轴对应到数据集里落到对应 Bin 的数据的数量。用 SQL 语言来说 y 就是 `count(*)` 。

第四场的分析

- 在工作中，我的确发现 pandas 的便利函数很方便，但是我得承认，脑子里总要惦记着 pandas 给箱线图和直方图提供了 `by` 参数，却没有给折线图提供该参数。
- 我把第一场和第二场分开是有原因的，其中最重要的原因是：从第二场开始 matplotlib 变得比较吓人。比如说要画箱线图还得记着用一个完全独立的界面，这根本不适合我。
- 说起第一场和第二场，有一个有趣的小细节：其实我一开始是因为 Seaborn 有丰富的「专利级」可视化函数（比如，distplot，小提琴图，回归图等等）而从 matplotlib/pandas 转移阵营的。但我后来喜欢上了 FacetGrid，我必须说这些第二场中的函数是 Seaborn 的杀手级应用。只要我还在画图我就离不开它们。
- （此外，我需要说明：Seaborn 提供了许多被小型库所忽略的优秀可视化函数；如果你碰巧需

射呈现给视图的方式就行了，比如换一下几何对象。

- 类似的，哪怕在第二场，Altair 的 API 也有非同寻常的一致性。哪怕对于那些看上去非常另类的操作，Altair 的 API 也非常简单、优雅，令人印象深刻。

数据说明

（在最后一幕，我们会处理「泰坦尼克」，另一个著名的整洁数据集（代码中仍然用 `df` 来表示）。下面是预览……）

survived	pclass	sex	age	fare
0	0	3	male	22.0
1	1	1	female	38.0
2	1	3	female	26.0
3	1	1	female	35.0
4	0	3	male	35.0

这个例子中，我们感兴趣的是看看每个客舱等级的平均费用是否和逃生率相关。显然，在 pandas 中我们可以这样写：

```
dfg = df.groupby(['survived', 'pclass']).agg({'fare': 'mean'})
dfg
```

survived	pclass	
0	1	64.684008
	2	19.412328
	3	13.669364
1	1	95.608029
	2	22.055700
	3	13.694887

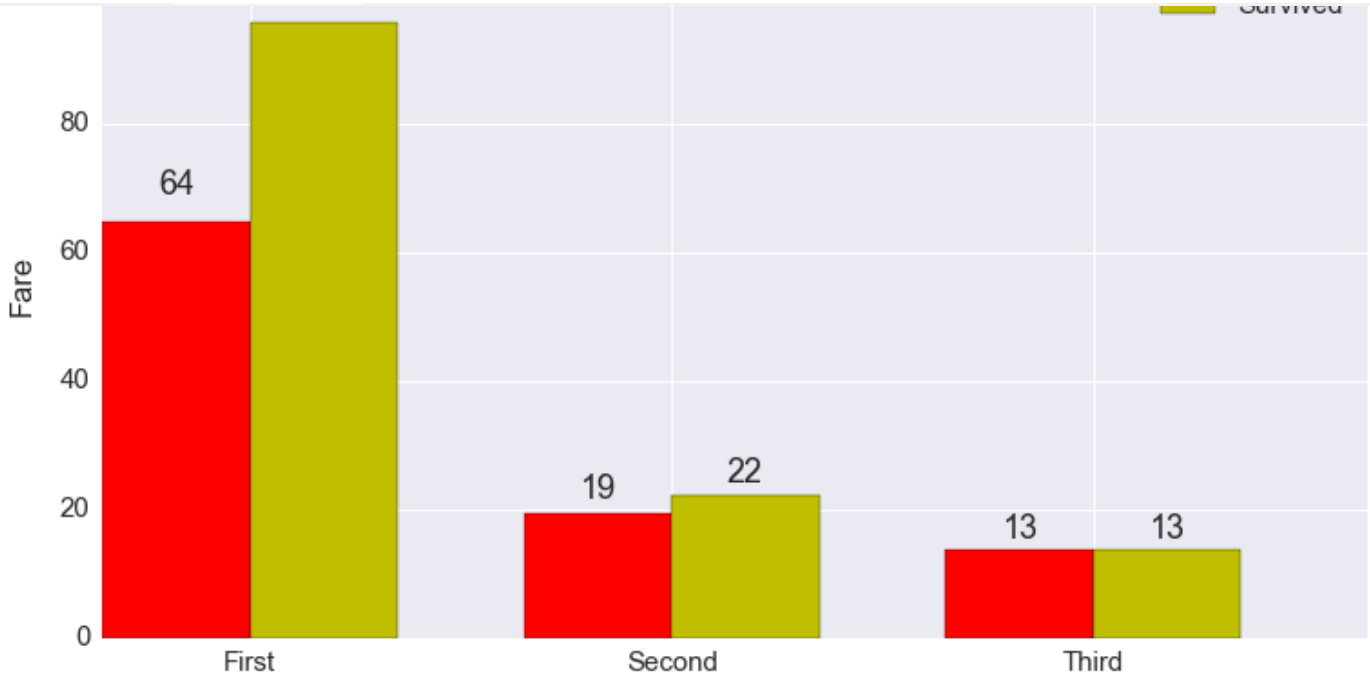
…… 不过这有什么意思呢？我写的可是数据可视化文章，所以用条形图再试一次！

第五场：如何画条形图？

MPL (表情严肃): 一句话也没说。

```
# MATPLOTLIB

died = dfg.loc[0, :]
survived = dfg.loc[1, :]

# more or less copied from matplotlib's own
# api example
fig, ax = plt.subplots(1, 1, figsize=(12.5, 7))

N = 3

ind = np.arange(N)  # the x locations for the groups
```

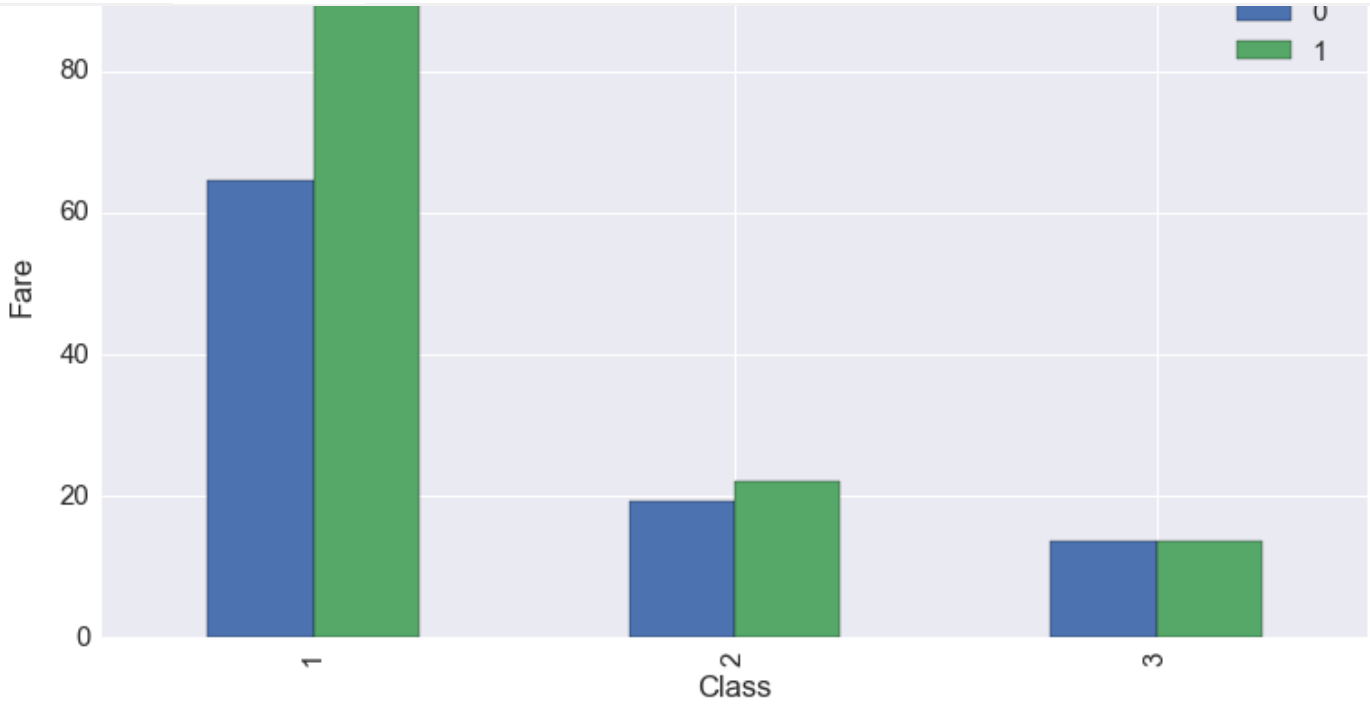
```
# add some text for labels, title and axes ticks
ax.set_ylabel('Fare')
ax.set_title('Fare by survival and class')
ax.set_xticks(ind + width)
ax.set_xticklabels(('First', 'Second', 'Third'))

ax.legend((rects1[0], rects2[0]), ('Died', 'Survived'))

def autolabel(rects):
    # attach some text labels
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,
                '%d' % int(height),
                ha='center', va='bottom')

ax.set_ylim(0, 110)

autolabel(rects1)
autolabel(rects2)

plt.show()
```



其他人都开始摇头

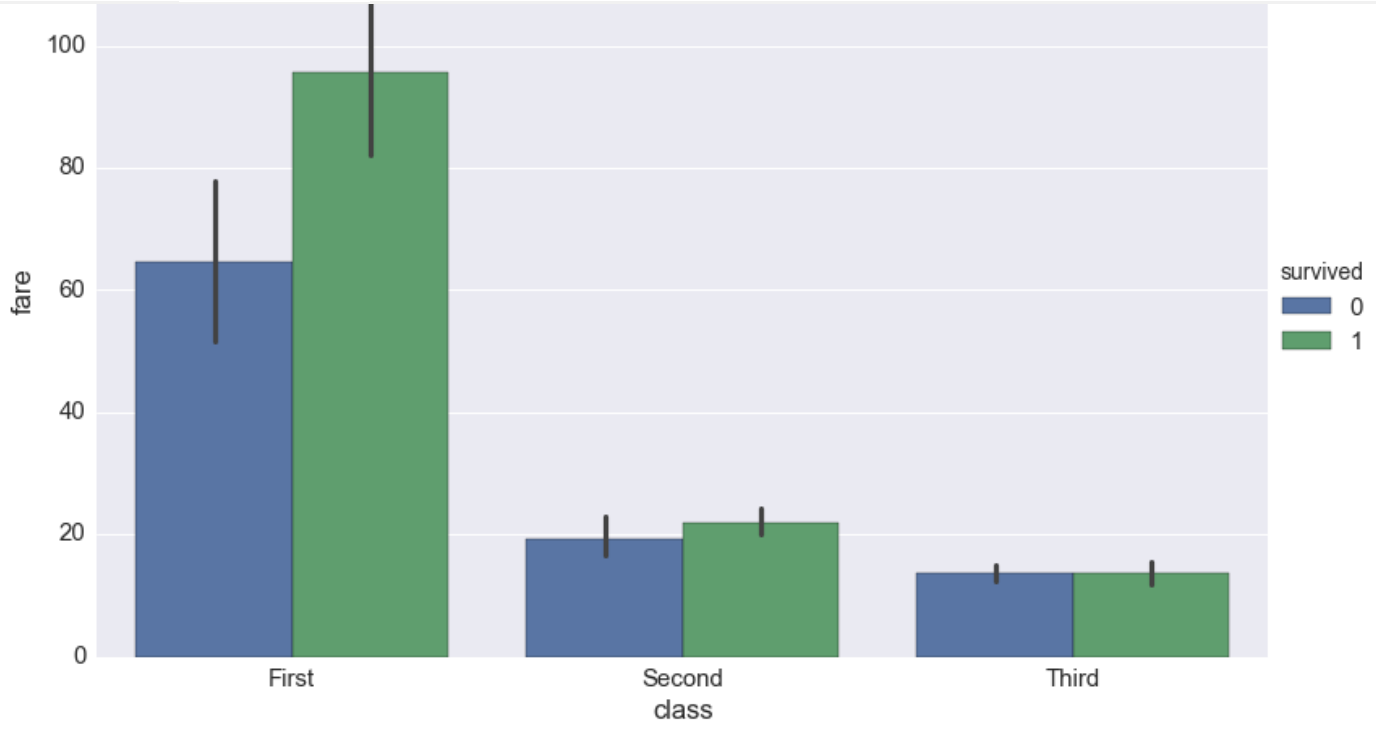
P: 我得先对数据进行一些处理 —— 也就是 `group by` 和 `pivot` —— 处理完就可以用非常帅气的条形图方法了，比上面这些简单得多！哇，我现在自信多了，我把其他人都比下去了！[5]

```
# PANDAS
fig, ax = plt.subplots(1, 1, figsize=(12.5, 7))
# note: dfg refers to grouped by
# version of df, presented above
dfg.reset_index().\
    pivot(index='pclass',
          columns='survived',
          values='fare').plot.bar(ax=ax)

ax.set(xlabel='Class',
       ylabel='Fare',
       title='Fare by survival and class')
```



SB: 我恰好又认为这类工作非常重要。鉴于此，我使用了特殊的 `factorplot` 函数来帮助我：

```
# SEABORN
g = sns.factorplot(x='class', y='fare', hue='survived',
                   data=df, kind='bar',
                   order=['First', 'Second', 'Third'],
                   size=7.5, aspect=1.5)
g.ax.set_title('Fare by survival and class')
```



SB: 跟之前一样，先将未处理过的数据传给数据框，再搞明白自己要按照什么进行分组，这里就是 `class` 和 `survived` ，它们对应 `x` 和 `hue` 变量。然后搞明白要对哪个数据列进行摘要统计，这里就是 `fare` ，对应到 `y` 变量。默认的摘要统计方法是求平均数，不过 `factorplot` 提供了 `estimator` 参数，可以通过它指定想要的函数，比如求和，标准差，中位数等等。而选择的函数会决定每个柱的高度。

当然，有很多方法可以可视化这个信息，条形图只有一种。同样我还提供了 `kind` 参数用来指定不同的可视化方法。

最后，**还有人**比较在意统计确定性，所以我会默认给你加上误差线，这样可以看出不同等级舱位的平均费用和生存率是否有关系。

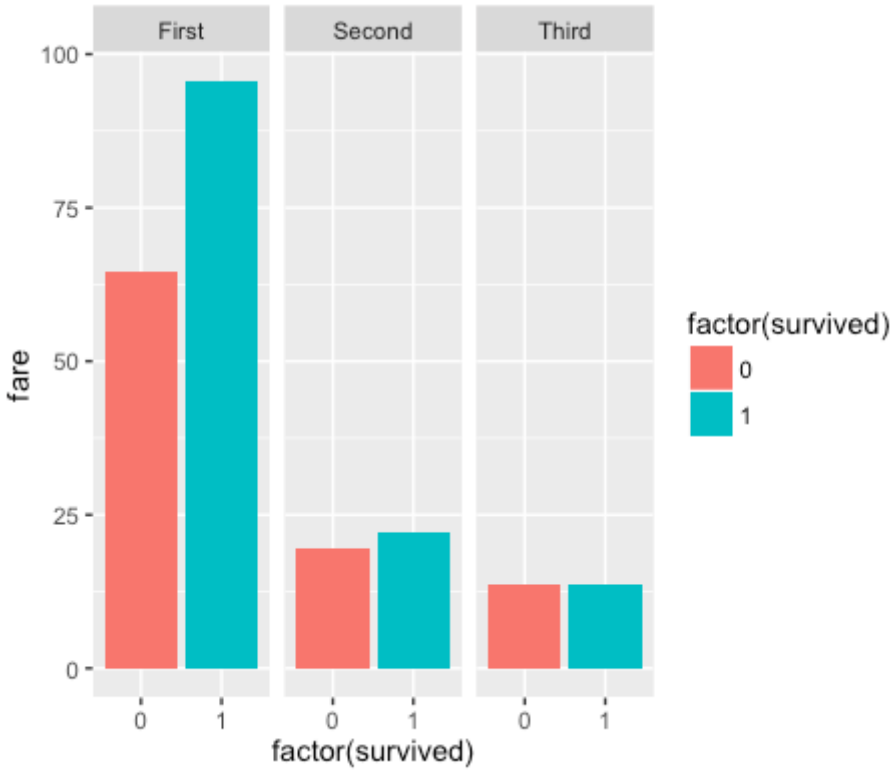
（**压低声音说**）希望你们做得比我还好

ggplot2 停下兰博基尼，走了进来

ggplo2: 嘿，你们看到 ——

GG: 嘿，大哥。

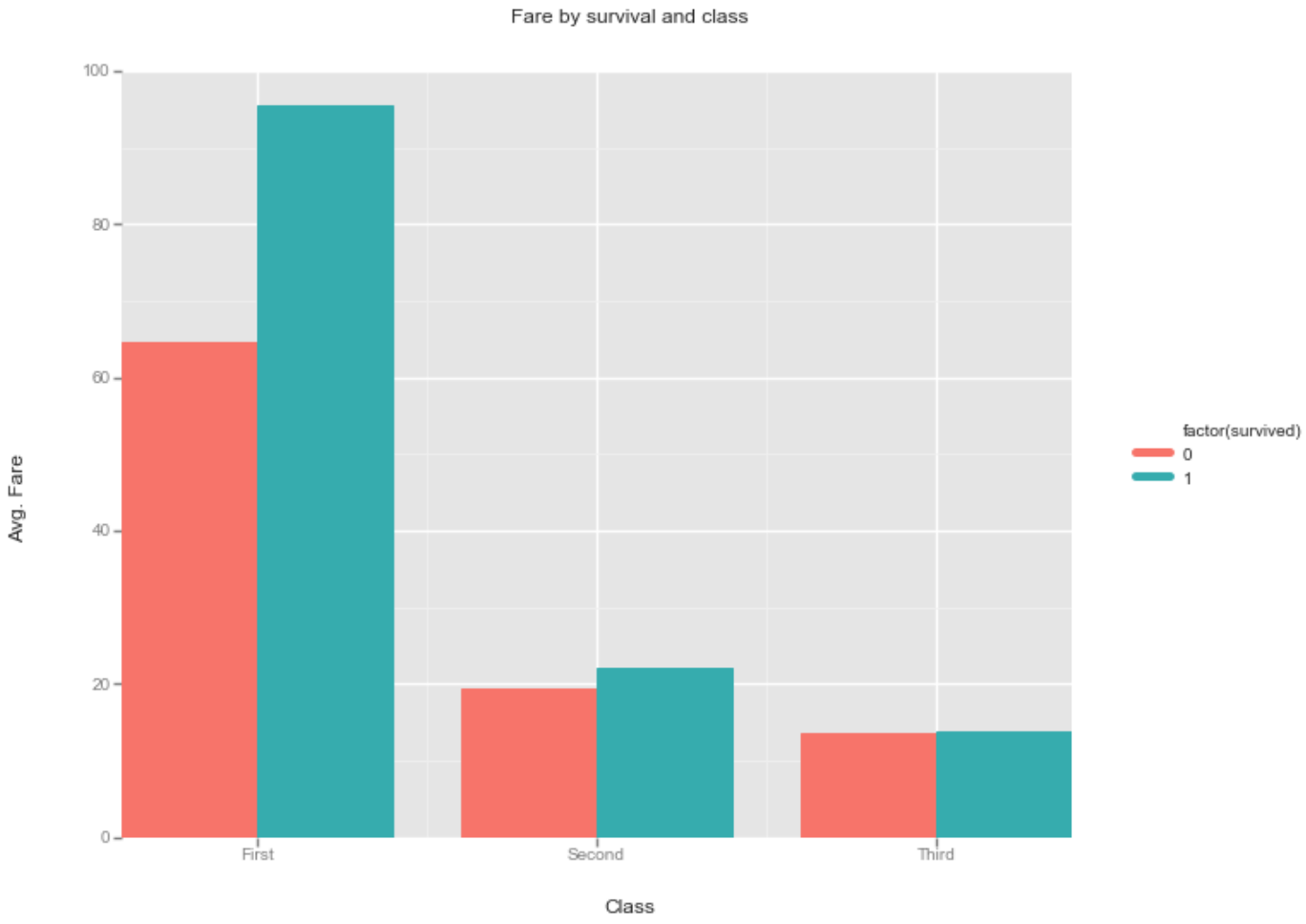
```
# GGPLOT2

# R 语言中你得这样写

ggplot(df, aes(x=factor(survived), y=fare)) +
    stat_summary_bin(aes(fill=factor(survived)),
                     fun.y=mean) +
    facet_wrap(~class)

# 天啊，ggplot2 可真棒
```



GG2: 看懂了吗？你要像我之前说的一样定义好图形映射，不过得把 `y` 映射到平均费用上。这就得叫我的好兄弟 `stat_summary_bin` 帮忙了，我只要把 `mean` 传给 `fun.y` 参数就行了。

GG（惊讶地睁大眼睛）： 哦，呃…… 我发现我还没有 `stat_summary_bin` 呢。我想想 —— pandas 你能帮帮我吗？

```
# GGPLOT
g = ggplot(df.groupby(['class', 'survived']).\
               agg({'fare': 'mean'}).\
               reset_index(), aes(x='class',
                                  fill='factor(survived)',
                                  weight='fare',
                                  y='fare')) + \
        geom_bar() + \
        ylab('Avg. Fare') + \
        xlab('Class') + \
        ggtitle('Fare by survival and class')
g
```



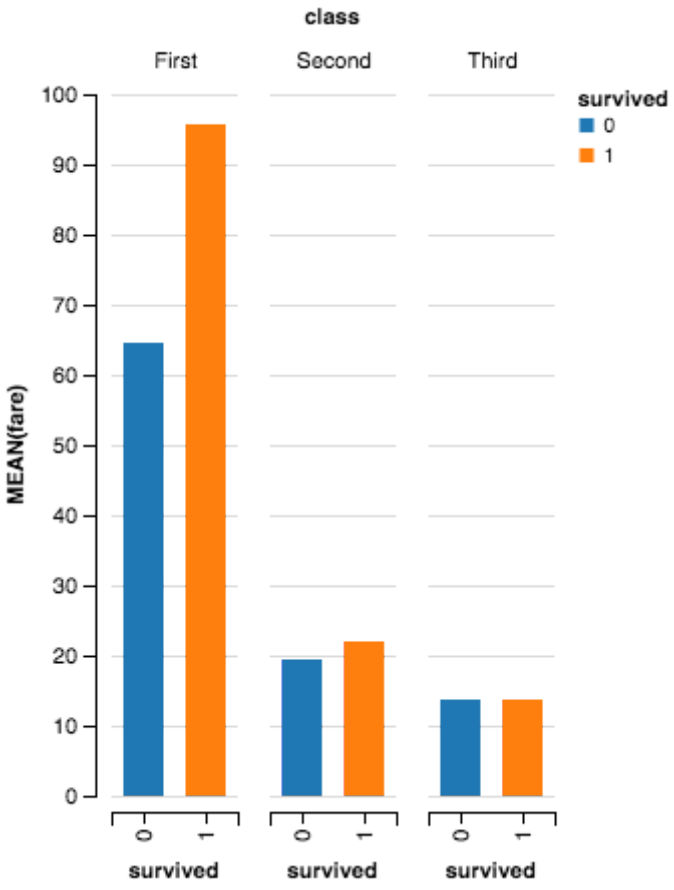
GG2: 噢，不完全是图形式语法，不过我觉得只要 Hadley 还没有发现，这样也能用…… 特别是你不应该在可视化之前就对数据进行汇总。我也不是特别懂这个上下文中 `weight` 是什么意思……

GG2: 噢，我懂了…… 我们以后再讨论吧。

GG 和 GG2 道别并离开了晚宴

ALT: 噢，现在**这**可是我的安身立命之道。非常简单。

```
# ALTAIR
c = Chart(df).mark_bar().encode(
    x='survived:N',
    y='mean(fare)',
    color='survived:N',
    column='class')
c.configure_facet_cell(strokeWidth=0, height=250)
```



ALT: 我希望下面的解释可以让所有的变量都非常直观：我想按幸存数来画平均船费，按舱位等级进行分面。写在代码里就是 `survived` 是 x 变量， `mean(fare)` 是 y 变量，而 `class` 是 column 变

`survived` 看上去像个定量变量，而定量变量有可能让绘的图变得有点丑。也不要太担心啦，这问题也不是每次都能碰到，只有个别情况下会有影响。比如，在上面的时间序列图中，如果我不知道 `dt` 是时间变量，我可能会假设它们只是名义变量，这样子就尴尬了（还好我在后面加上了 `:T` ，这样就好了）。

另外我还用了 `configure_facet_cell` 协议让三个子图看上去更加统一。

第五场的分析

- 这条不要想太多：我再也不用 matplotlib 画条形图了，明确地说，这不是我的个人观点！事实上 matplotlib 不会像其他的库那样对传入的数据进行推测。这有时候就意味着你得写严格的命令式代码。
- （当然，正是这种数据不可知论让 matplotlib 成了其他 Python 可视化库的基础。）
- 相对而言，在需要汇总统计和误差线的时候，我总是会用 Seaborn。
- （这样比较可能有失公允，毕竟我选的例子仿佛是为 Seaborn 的一个函数量身定制的，不过我的工作中这种事遇到的太多了，而且，嘿，这篇文章可是我写的。）
- 我不觉得 pandas 或 ggplot 的方式有什么特别的优势。
- 不过，就 pandas 而言，哪怕是简单的条形图也必须得记得用 `group by` 和 `pivot` ，这看上去有点傻。
- 同样，我的确认为这是 yhat 开发的 ggplot 的一个重大缺陷，要找一个 `stat_summary` 的替代品从而让 ggplot 变得功能完善全面还有很长的路要走。
- 同时，Altair 依然让我印象深刻！我被解决这个例子的代码的直观性震惊了。哪怕你从来没有见过 Altair，我也能想象有人是可以看懂的。正是它**这种**思考，代码和可视化的一一对应让它成了我最爱的库。

最后的感想

（是啊，这只是种逃避。）

尽管我在 matplotlib 上遇到了点困难，它还是非常好玩的（每一部剧都要有搞笑的部分）。不仅仅是因为 matplotlib 是 pandas，Seaborn 和 ggplot 这些库的底层基础，而且是因为它给予你非常细粒度的控制权。虽然我没有说，但是我用 matplotlib 调整了所有非 Altair 所绘的图。但是，注意听，matplotlib 是纯声明式的，非常细节地指定可视化图像的方方面面简直是无趣的（看看条形图的例子吧）。

的确，还有这种结果：「用统计可视化能力来评价 matplotlib 是不公平的，你这个刻薄的家伙。你在用它的一种使用案例来和其他库的主要使用案例进行比较。这些方法显然应该一起使用。你可以使用自己喜欢的方便的/陈述式表达层—— pandas，Seaborn，ggplot 或者将来的 Altair（下文会详述）—— 来做基础工作。然后用 matplotlib 完成那些非基础工作。如果你穷尽了其他库所能提供的一切也找不到想要的东西，你会很高兴可以看到能力无限的 matplotlib 就在你身边，你这个不知感恩的业余绘图的。」

对于这些人我得说：是的！这很有道理，却脱离现实……，尽管只是**说**这些并不会撑起博文的大部分内容。

再说，要是我就不会骂人。

同时，轴向旋转（pivot）结合 pandas 处理时间序列图像非常好用。考虑到 pandas 的时间序列支持更加广泛，我还会接着用。此外，下一次如果要画 RadViz 图，我就知道该怎么做了。也就是说，尽管 pandas 的确在 matplotlib 的命令式范式的基础上提供了声明式语法（比如条形图），它仍然极具 matplotlib 风格。

接着说：如果你想要一些更偏向统计的东西，用 Seaborn 吧（她的确在国外学到了很多很酷的东西）。学习她的 API —— factorplot, regplot, displot 等等等等 —— 然后爱上她。这时间花得值。至于 faceting，我觉得 FacetGrid 是个很有用的共犯（wtf！）；但是要不是我使用 Seaborn 已久，我可能更喜欢 ggplot 或 Altair。

说到声明式的优雅，我一直深爱着 ggplot2 ，而且对 Python 的 ggplot 留下了深刻印象。我肯定会持续关注这个项目。（更自私地说，我希望它可以阻止那些使用 R 语言同事取笑我。）

最后，如果你要做的事可以用 Altair 完成（抱歉了，箱线图使用者），用它吧！它提供的 API 异常简单又非常好用。如果还需要其他动力，想想这些：Altair 一个令人激动的特性是（除了即将到来

这有什么好激动的？好吧，在底层，所有的可视化看上去都是这个样子的：

```
    {'dt': '2002-09-19', 'kind': 'A', 'value': -6.403886978049999},
    {'dt': '2002-09-20', 'kind': 'A', 'value': -6.897465243919999},
    {'dt': '2002-09-21', 'kind': 'A', 'value': -5.96757445979},
    {'dt': '2002-09-22', 'kind': 'A', 'value': -5.89114640739},
    {'dt': '2002-09-23', 'kind': 'A', 'value': -4.228217235530001},
    {'dt': '2002-09-24', 'kind': 'A', 'value': -5.6031405023800005},
    {'dt': '2002-09-25', 'kind': 'A', 'value': -7.110818410760001},
    {'dt': '2002-09-26', 'kind': 'A', 'value': -7.238857028310001},
    ...]},
 'encoding': {'color': {'field': u'kind', 'type': 'nominal'},
  'x': {'field': u'dt', 'type': 'temporal'},
  'y': {'field': u'value', 'type': 'quantitative'}},
 'mark': 'line'}
```

的确，看上去没什么好激动的，但是想想它的影响：如果其他的库对此感兴趣，他们可以直接开发新方法将这些 Vega-Lite JSON 对象转换成可视化结果。这就意味着可以用 Altair 搞定基本工作，然后深入底层用 matplotlib 获得更多控制。

我已经对此期待万分了。

说完这一切，再说几句告别的话：Python 可视化可比一个男人，女人或者尼斯湖水怪大多了。所以你得有选择地接受我刚才说的一切，不论是代码还是意见。记得：互联网上的一切都是谎言，该死的谎言和统计。

希望你喜欢这个书呆子气十足的疯帽匠茶会，如果学到了什么东西你可以用到自己的工作中。

照旧, 代码在 GitHub 上。

注释

首先，非常感谢订阅了 /u/counters 的 reddit 用户，你们在这个评论留下了非常有价值的反馈和观点。我选取了一些放在了「最后的感谢」一节；不过我的表示远没有那么清楚，也就是说，看看那个评论吧；非常不错。

其次，非常非常感谢 Thomas Caswell 他写的关于 matplotlib 的特性的评论你绝对要读一读。

2. 马上解释一下，你都对我愤怒了，所以允许我解释一二：我爱 bokeh 和 plotly。真的，我在提交分析之前最爱做的一件事就是把图像传给相关的 bokeh/plotly 函数，获得自由的交互性；但是我对它俩都不是特别熟，没法做更高级的操作。（说实话，这篇文章已经够长的了。）

显然，如果你要的是交互可视化（而不是统计可视化），你可能就得找它俩了。

1. **请**注意：这只是为了好玩。我**没有**用业余的拟人化手法评价任何库。我相信显示生活中的 matplotlib 是非常可爱的。

2. 坦率地说，我不是**完全**确定单独进行分面操作是为了意识形态上的纯洁，或者只是单纯出于实用的考虑。虽然我的 ggplot 角色声称他是前者（他的理解来自匆匆读完的这篇论文），也有可能是因为（实际上）ggplot2 对分面的支持太丰富了，所以需要当作是独立的步骤。如果我描述的角色违反了任何图形语法规则，请务必告诉我，我会去找个新的。

3. 绝对不是这个故事的道德准则。

Python 数据可视化