Plotting and Visualization

Part 4

Plotting with pandas and seaborn

Part 1

- matplotlib can be a fairly low-level tool.
- You assemble a plot from its base components: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations.

- In pandas we may have multiple columns of data, along with row and column labels.
- pandas itself has built-in methods that simplify creating visualizations from DataFrame and Series objects.
- Another library is `seaborn`, a statistical graphics library created by Michael Waskom.
- Seaborn simplifies creating many common visualization types.

Line Plots

- Series and DataFrame each have a `plot` attribute for making some basic plot types.
- By default, `plot()` makes line plots:

```
In [28]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
         s.plot()
```

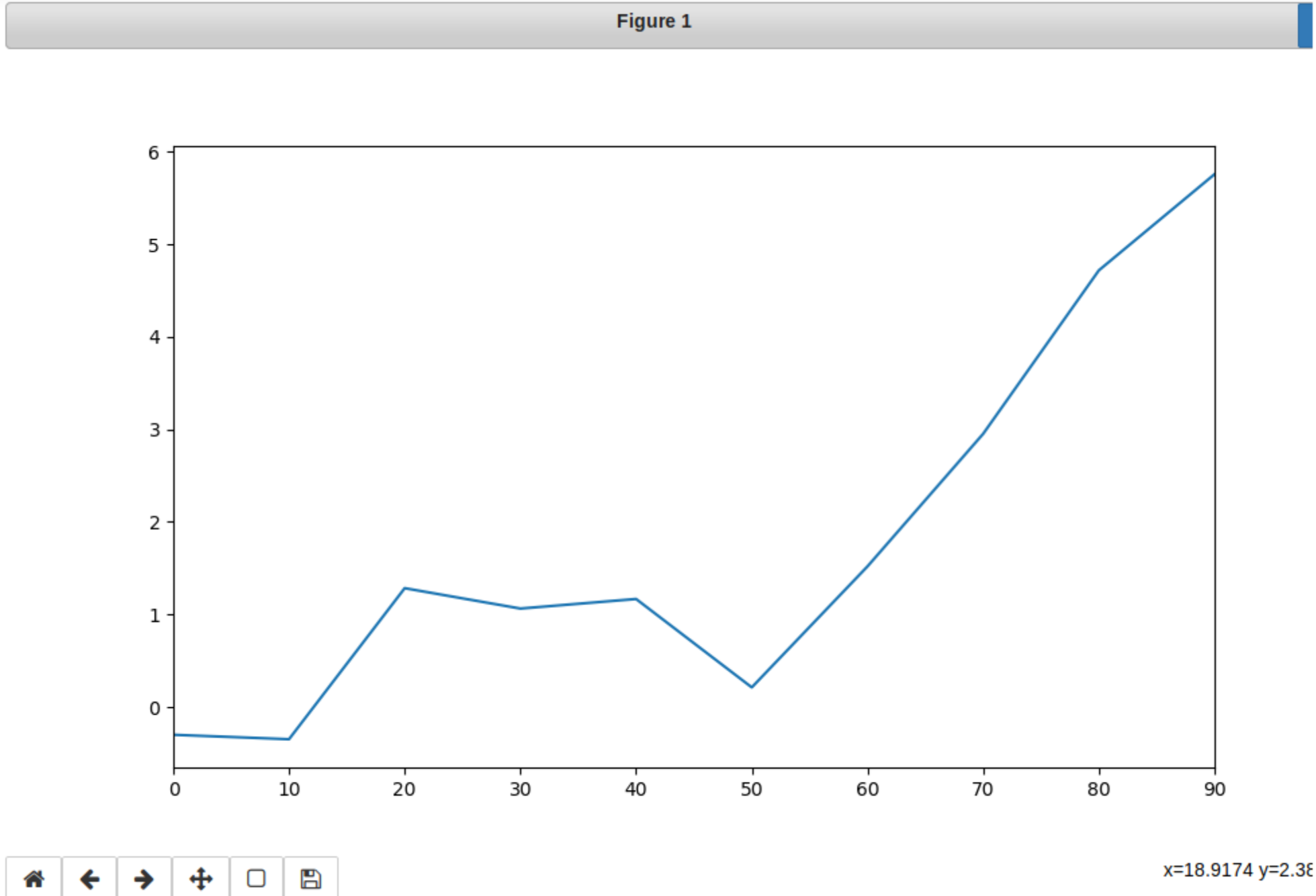


Figure 1
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
90
x=18.9174 y=2.38

- Most of pandas's plotting methods accept an optional `ax` parameter, which can be a matplotlib subplot object.
- This gives you more flexible placement of subplots in a grid layout.

- DataFrame's `plot` method plots each of its columns as a different line on the same subplot, creating a legend:

```
In [29]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
                           columns=['A', 'B', 'C', 'D'],
                           index=np.arange(0, 100, 10))
         df.plot()
```

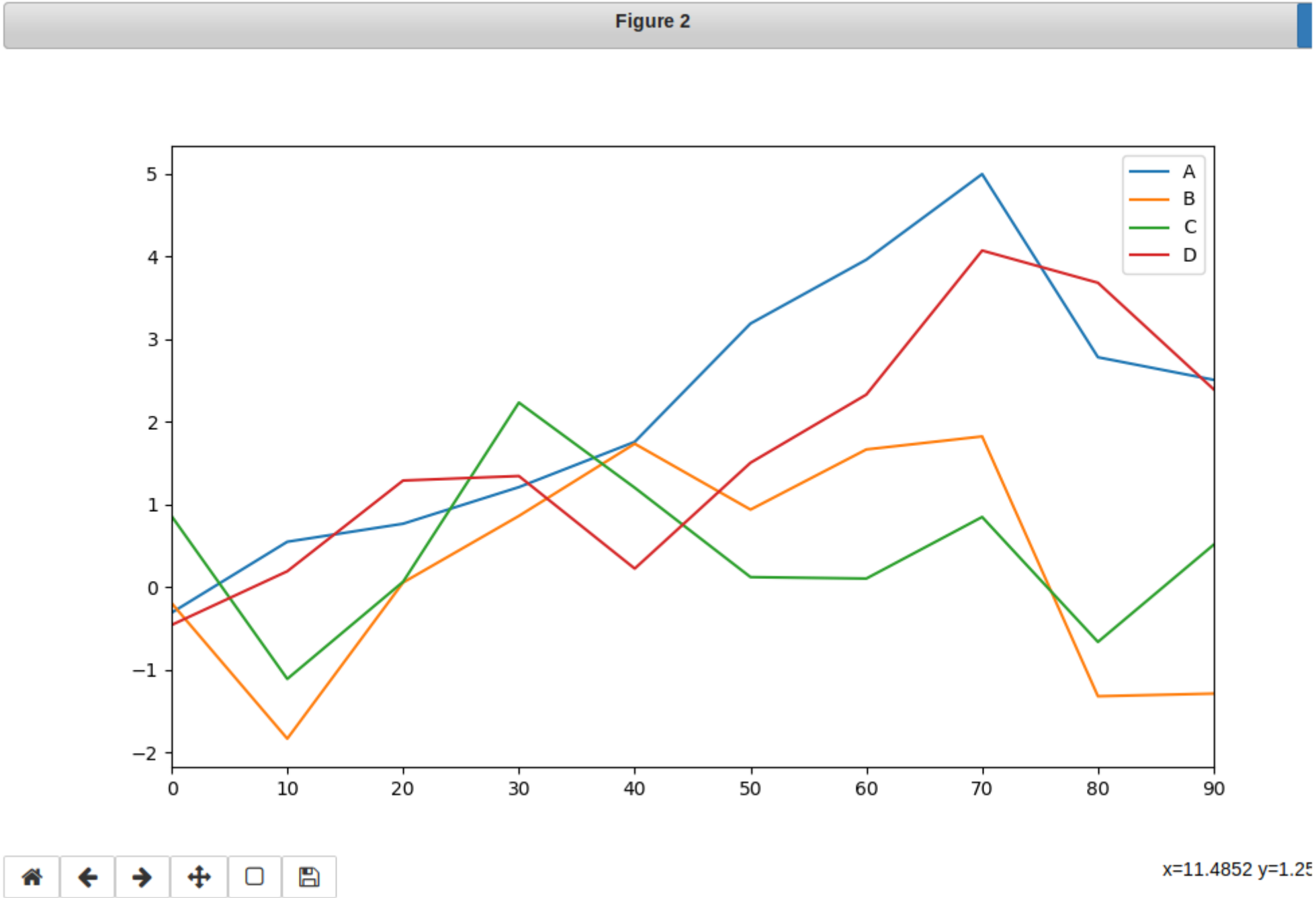


Figure 2
A
B
C
D
5
4
3
2
1
0
−1
−2
0
10
20
30
40
50
60
70
80
90
x=11.4852 y=1.25

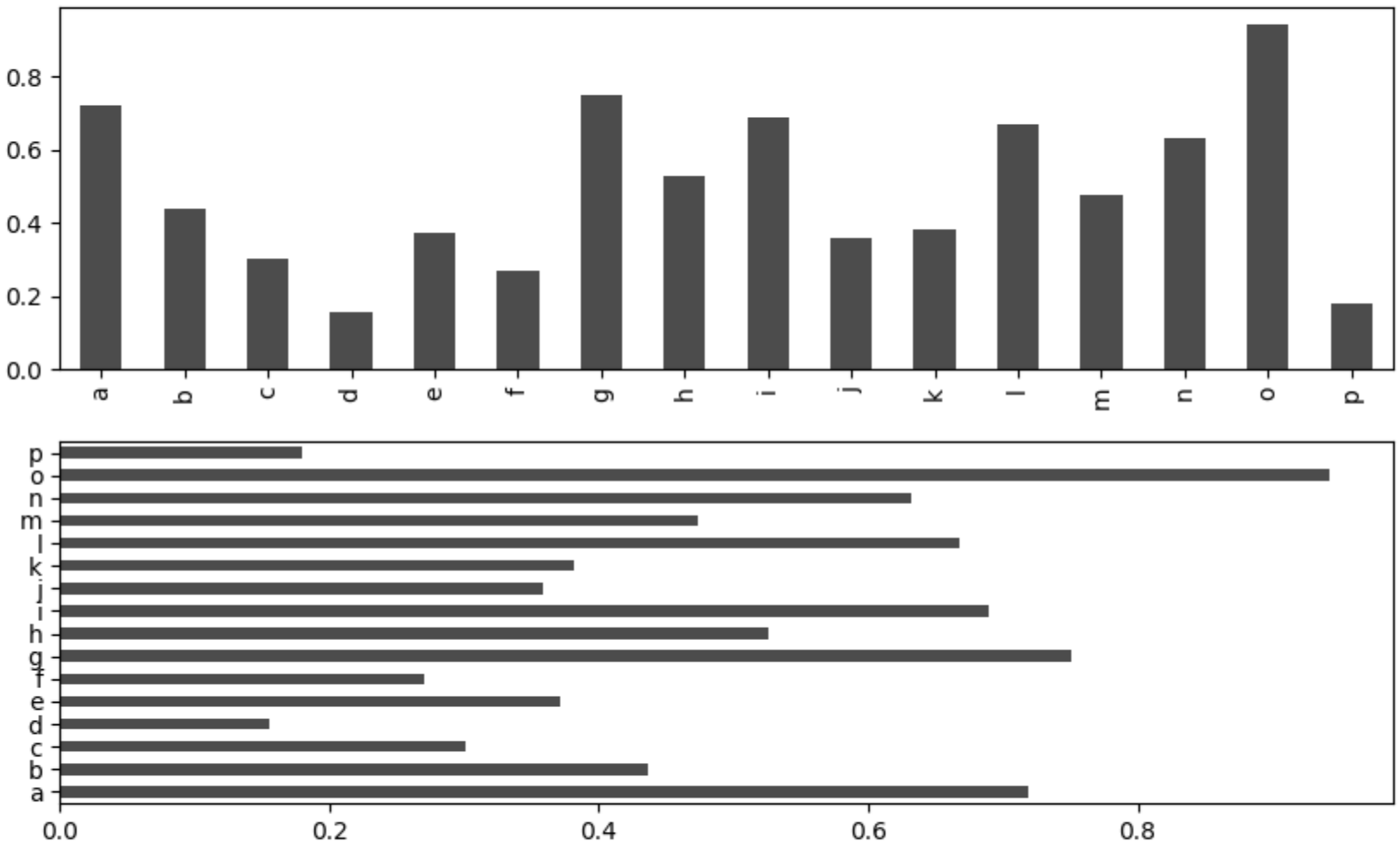
- The `plot` attribute contains a “family” of methods for different plot types.
- For example, `df.plot()` is equivalent to `df.plot.line()`.

Bar Plots

- The `plot.bar()` and `plot.barh()` make vertical and horizontal bar plots, respectively.
- In this case, the Series or DataFrame index will be used as the x (`bar`) or y (`barh`) ticks:

```
In [30]: fig, axes = plt.subplots(2, 1)
         data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
         data.plot.bar(ax=axes[0], color='k', alpha=0.7)
         data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

Figure 3

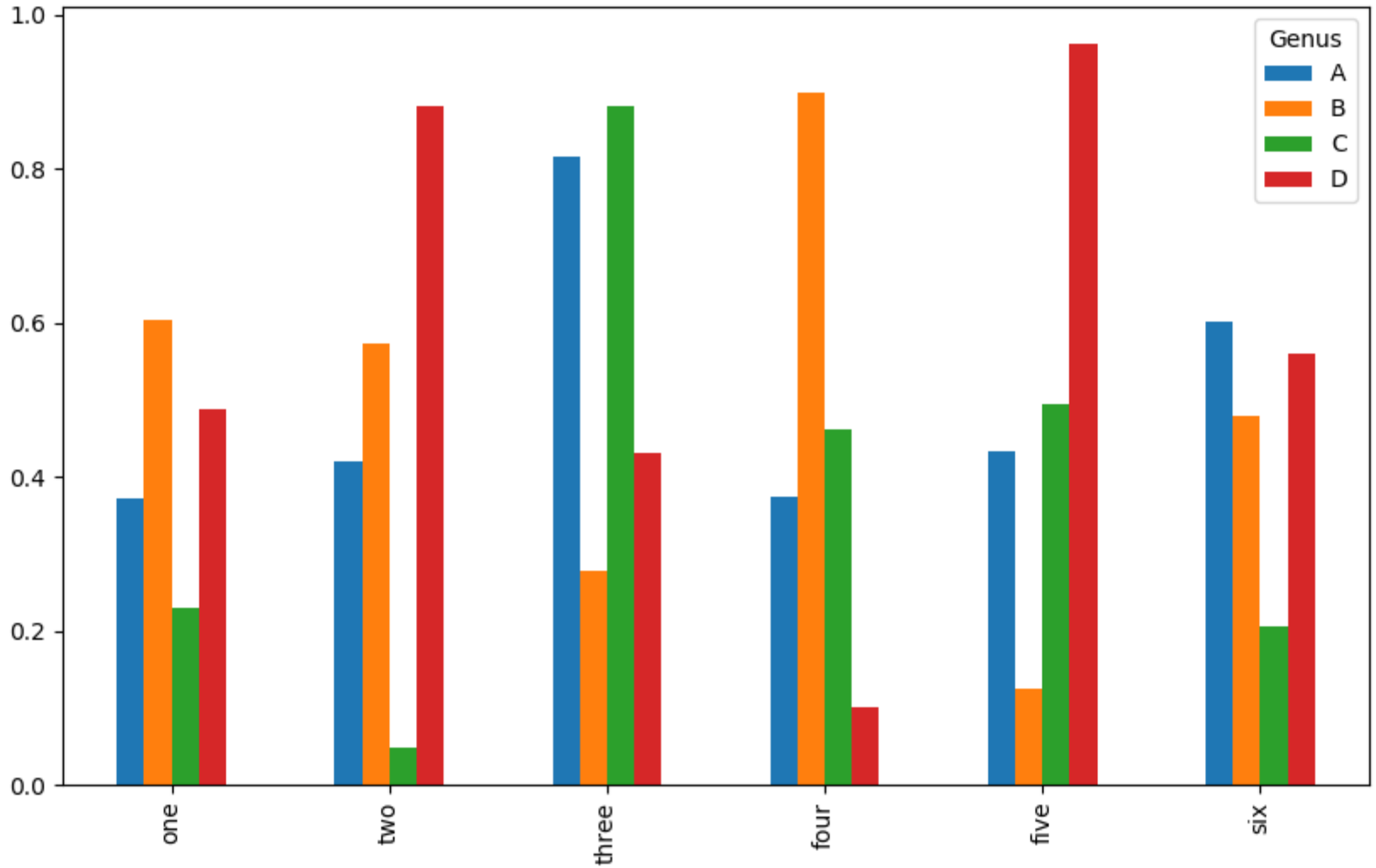


x= y=0.0537

- With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value.

```
In [32]: df = pd.DataFrame(np.random.rand(6, 4),
                           index=['one', 'two', 'three', 'four', 'five', 'six'],
                           columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
         df
         df.plot.bar()
```

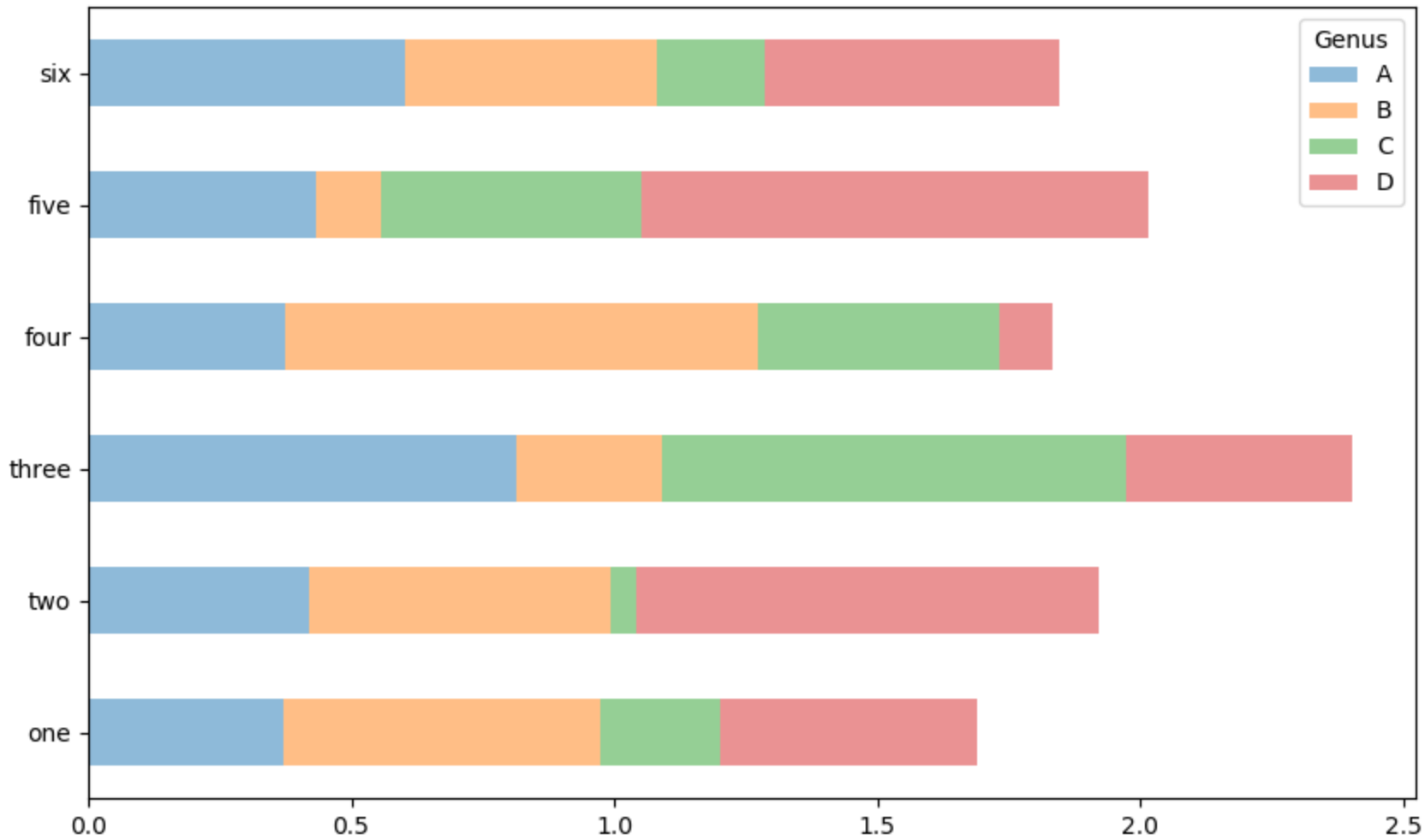
Figure 4



- We create stacked bar plots from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together:

```
In [35]: df.plot.barh(stacked=True, alpha=0.5)
```

Figure 6



- A useful recipe for bar plots is to visualize a Series's value frequency using `value_counts`: `s.value_counts().plot.bar()`.

- Consider the tipping dataset, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day.
- We load the data using `read_csv` and make a cross-tabulation by day and party size:

In [38]:
```
tips = pd.read_csv('examples/tips.csv')
party_counts = pd.crosstab(tips['day'], tips['size'])
party_counts
```

Out[38]:

size	1	2	3	4	5	6
day						
Fri	1	16	1	1	0	0
Sat	2	53	18	13	1	0
Sun	0	39	15	18	3	1
Thur	1	48	4	5	1	3

In [39]:
```
# Not many 1- and 6-person parties
party_counts = party_counts.loc[:, 2:5]
```

- Then, normalize so that each row sums to 1 and make the plot:

In [41]:
```
# Normalize to sum to 1
party_pcts = party_counts.div(party_counts.sum(1), axis=0)
party_pcts
```

Out[41]:

size	2	3	4	5
day				
Fri	0.888889	0.055556	0.055556	0.000000
Sat	0.623529	0.211765	0.152941	0.011765
Sun	0.520000	0.200000	0.240000	0.040000
Thur	0.827586	0.068966	0.086207	0.017241

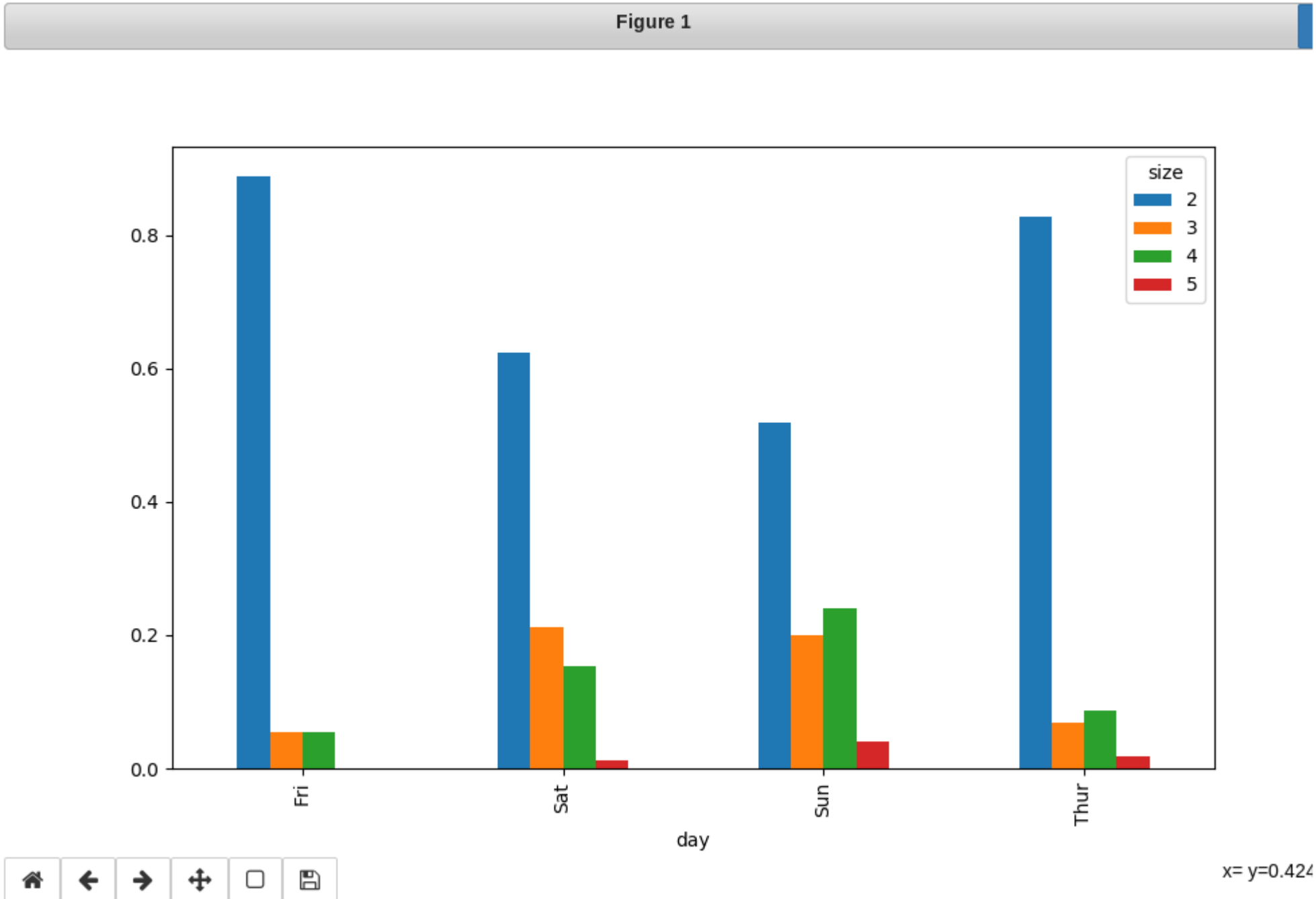


Figure 1
size
2
3
4
5
0.8
0.6
0.4
0.2
0.0
Fri
Sat
Sun
Thur
day
x= y=0.424

- With data that requires aggregation or summarization before making a plot, using the `seaborn` package can make things much simpler.
- Let's look now at the tipping percentage by day with `seaborn`:

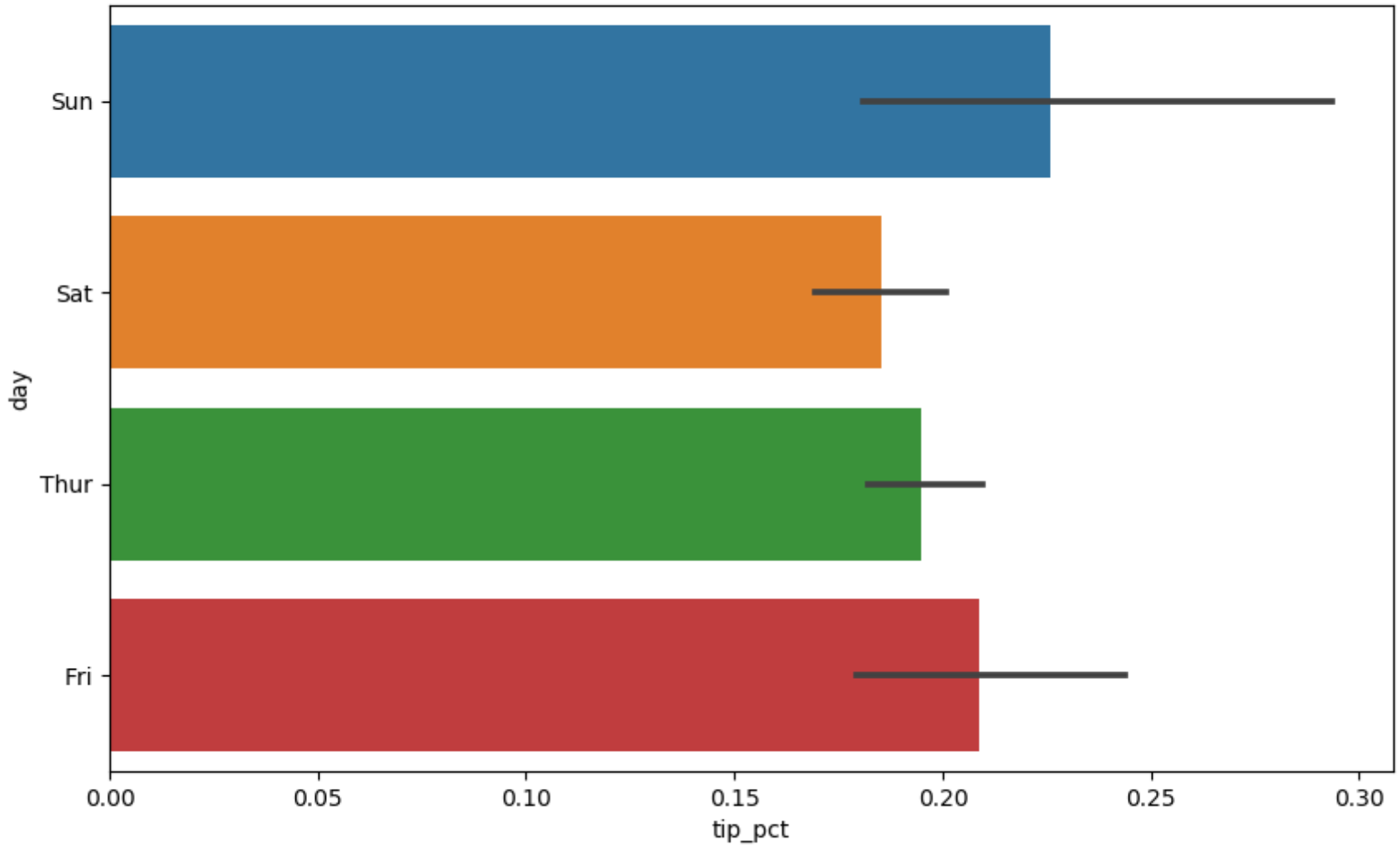
In [44]:
```
import seaborn as sns
tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])
tips.head()
```

Out[44]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.063204
1	10.34	1.66	No	Sun	Dinner	3	0.191244
2	21.01	3.50	No	Sun	Dinner	3	0.199886
3	23.68	3.31	No	Sun	Dinner	2	0.162494
4	24.59	3.61	No	Sun	Dinner	4	0.172069

In [45]:
```
sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

Figure 1



- Plotting functions in seaborn take a `data` argument, which can be a pandas DataFrame.
- The other arguments refer to column names.
- Because there are multiple observations for each value in the day, the bars are the average value of `tip_pct`.
- The black lines drawn on the bars represent the 95% confidence interval (this can be configured through optional arguments).

- `seaborn.barplot` has a `hue` option that enables us to split by an additional categorical value:

```
In [47]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

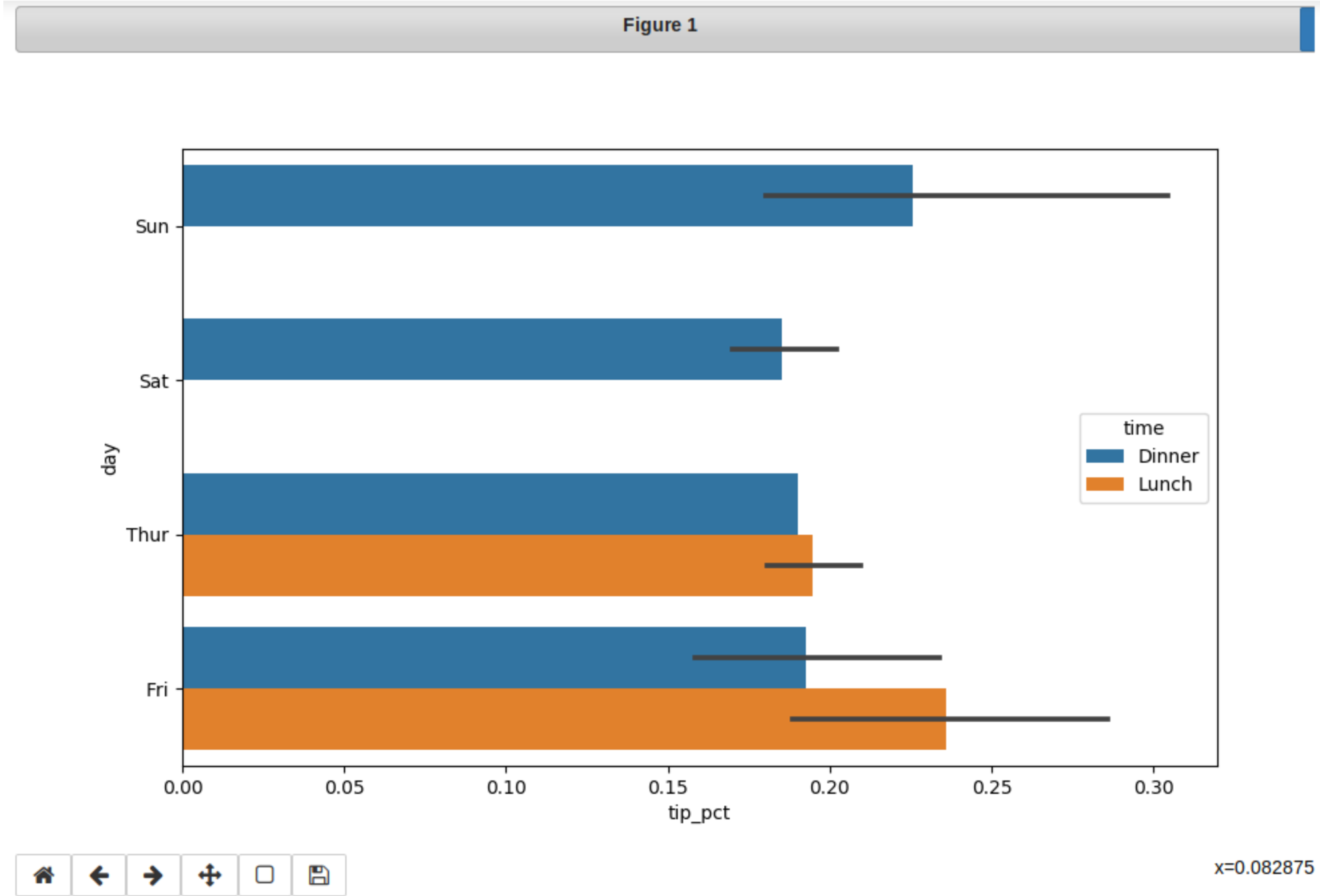


Figure 1
Sun
Sat
Thur
Fri
day
time
Dinner
Lunch
0.00
0.05
0.10
0.15
0.20
0.25
0.30
tip_pct
x=0.082875

- Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors.
- You can switch between different plot appearances using `seaborn.set`:

```
In [51]: sns.set(style="whitegrid")

In [52]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

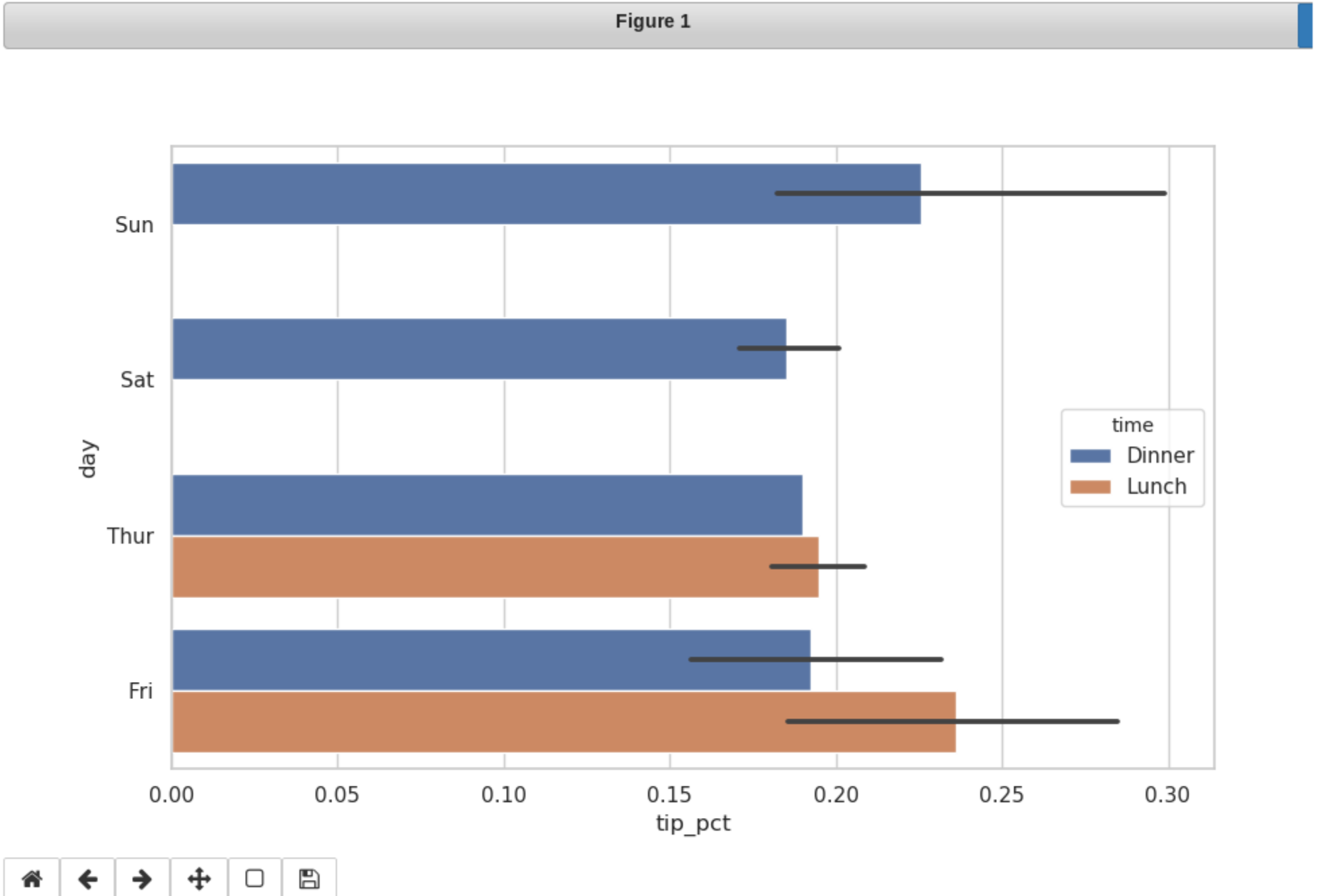


Figure 1
Sun
Sat
Thur
Fri
day
time
Dinner
Lunch
0.00
0.05
0.10
0.15
0.20
0.25
0.30
tip_pct