```
In[4]: fig, ax = plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax)

       # Add labels to the plot
       style = dict(size=10, color='gray')

       ax.text('2012-1-1', 3950, "New Year's Day", **style)
       ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
       ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
       ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
       ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
       ax.text('2012-12-25', 3850, "Christmas ", ha='right', **style)

       # Label the axes
       ax.set(title='USA births by day of year (1969-1988)',
              ylabel='average daily births')

       # Format the x axis with centered month labels
       ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
       ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
       ax.xaxis.set_major_formatter(plt.NullFormatter())
       ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```

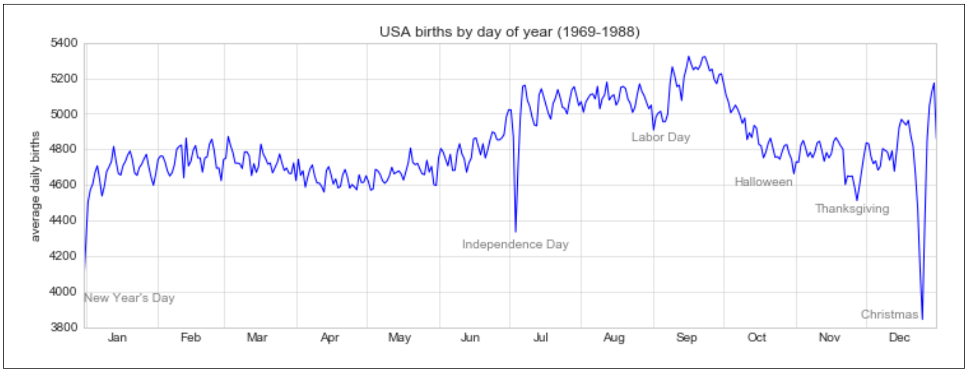

Figure 4-68. Annotated average daily births by date

The `ax.text` method takes an *x* position, a *y* position, a string, and then optional keywords specifying the color, size, style, alignment, and other properties of the text. Here we used `ha='right'` and `ha='center'`, where `ha` is short for *horizonal alignment*. See the docstring of `plt.text()` and of `mpl.text.Text()` for more information on available options.

Transforms and Text Position

In the previous example, we anchored our text annotations to data locations. Sometimes it's preferable to anchor the text to a position on the axes or figure, independent of the data. In Matplotlib, we do this by modifying the *transform*.

Any graphics display framework needs some scheme for translating between coordinate systems. For example, a data point at $(x, y) = (1, 1)$ needs to somehow be represented at a certain location on the figure, which in turn needs to be represented in pixels on the screen. Mathematically, such coordinate transformations are relatively straightforward, and Matplotlib has a well-developed set of tools that it uses internally to perform them (the tools can be explored in the `matplotlib.transforms` submodule).

The average user rarely needs to worry about the details of these transforms, but it is helpful knowledge to have when considering the placement of text on a figure. There are three predefined transforms that can be useful in this situation:

`ax.transData`
: Transform associated with data coordinates

`ax.transAxes`
: Transform associated with the axes (in units of axes dimensions)

`fig.transFigure`
: Transform associated with the figure (in units of figure dimensions)

Here let's look at an example of drawing text at various locations using these transforms (Figure 4-69):

```
In[5]: fig, ax = plt.subplots(facecolor='lightgray')
       ax.axis([0, 10, 0, 10])

       # transform=ax.transData is the default, but we'll specify it anyway
       ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
       ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
       ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```

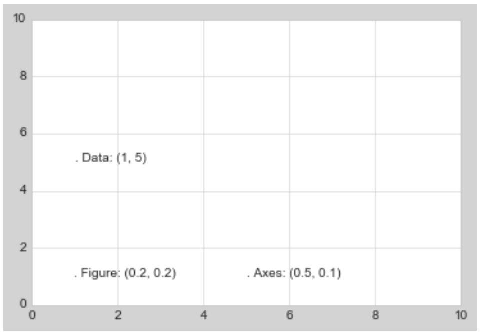

Figure 4-69. Comparing Matplotlib's coordinate systems

Note that by default, the text is aligned above and to the left of the specified coordinates; here the "." at the beginning of each string will approximately mark the given coordinate location.

The `transData` coordinates give the usual data coordinates associated with the x- and y-axis labels. The `transAxes` coordinates give the location from the bottom-left corner of the axes (here the white box) as a fraction of the axes size. The `transFigure` coordinates are similar, but specify the position from the bottom left of the figure (here the gray box) as a fraction of the figure size.

Notice now that if we change the axes limits, it is only the `transData` coordinates that will be affected, while the others remain stationary (Figure 4-70):

```
In[6]: ax.set_xlim(0, 2)
       ax.set_ylim(-6, 6)
       fig
```

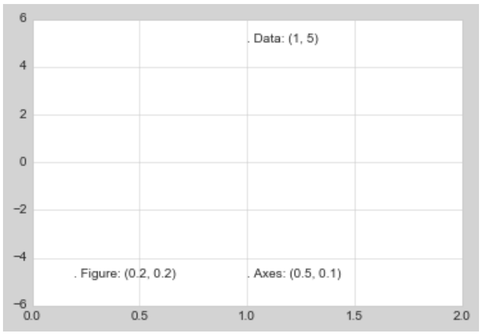

Figure 4-70. Comparing Matplotlib's coordinate systems

You can see this behavior more clearly by changing the axes limits interactively; if you are executing this code in a notebook, you can make that happen by changing `%matplotlib inline` to `%matplotlib notebook` and using each plot's menu to interact with the plot.

Arrows and Annotation

Along with tick marks and text, another useful annotation mark is the simple arrow.

Drawing arrows in Matplotlib is often much harder than you might hope. While there is a `plt.arrow()` function available, I wouldn't suggest using it; the arrows it creates are SVG objects that will be subject to the varying aspect ratio of your plots, and the result is rarely what the user intended. Instead, I'd suggest using the `plt.annotate()` function. This function creates some text and an arrow, and the arrows can be very flexibly specified.