**A Python Guide to Dynamic Chart Visualizations**

[[Nicolás Echeverry - Research Economist](https://medium.com/@necheverry_20468?source=post_page-----19aa4624fe10--------------------------------)](https://medium.com/@necheverry_20468?source=post_page-----19aa4624fe10--------------------------------)

[Nicolás Echeverry - Research Economist](https://medium.com/@necheverry_20468?source=post_page-----19aa4624fe10--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F7b082a4caa7e&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40necheverry_20468%2Fimprove-your-visualizations-with-animated-charts-in-python-19aa4624fe10&user=Nicol%C3%A1s+Echeverry+-+Research+Economist&userId=7b082a4caa7e&source=post_page-7b082a4caa7e----19aa4624fe10---------------------post_header-----------)

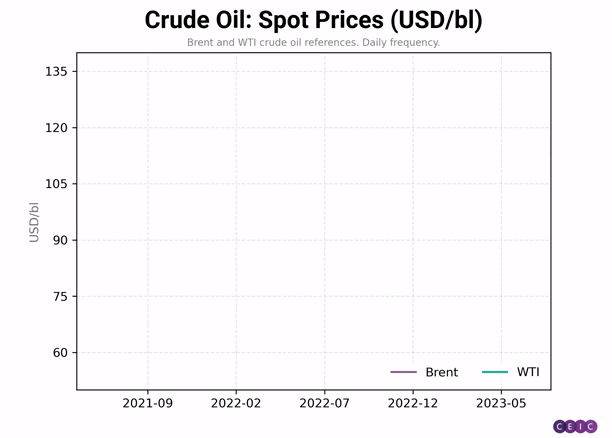
7 min read

·

Oct 4

30

2



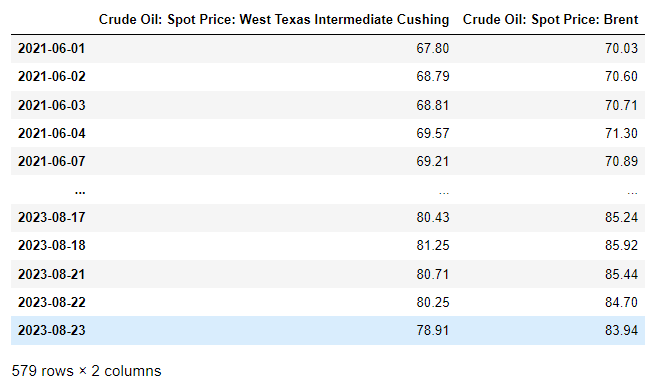
Data, when presented right, can captivate, educate, and inspire. While traditional charts provide a snapshot, animated visualizations offer a dynamic journey, making complex ideas more digestible and engaging.

In this article, we’ll explore the dynamic world of animated chart creation using Python, drawing upon real-world macroeconomic data as our canvas. We’ll guide you through the essentials of animating line charts and offer tips to refine your visuals, ensuring you present data like an expert. If you want to dive deeper, the complete code awaits in a [Jupyter Notebook](https://github.com/necheverryisi/CEIC-Data-Insights/blob/main/How%20to%20Animate%20Line%20Plot%20in%20Python.ipynb) in our [GitHub repository.](https://github.com/necheverryisi/CEIC-Data-Insights)

**Data**

Let’s talk about the data we are going to use. At [CEIC Data](https://www.ceicdata.com/en) we specialize in macroeconomic series of more than 200 economies. We’ll be using two well-known indicators: Brent and WTI crude oil daily prices from the U.S. Energy Information Administration.

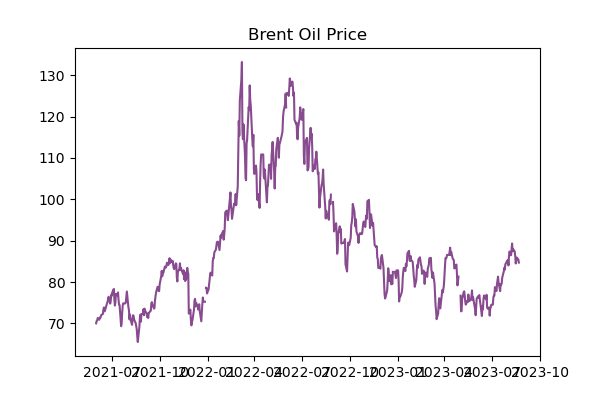
These indicators are often followed as oil is the world’s largest commodity market and its price behavior gives important signals about the global economy. Our dataset looks like this:



**Visualization**

Now that we have our data let’s start with a simple line plot that we will use as the base for the animation. Let’s create a figure and a line chart using the Matplotlib package from Python:

# Create figure and axes object  
fig, ax = plt.subplots(figsize=(6, 4))  
  
# Set title  
ax.set\_title('Brent Oil Price')  
  
# Plot the line  
ax.plot(oil\_df.index, oil\_df.iloc[:,1], color= '#884b8f')  
  
plt.show()



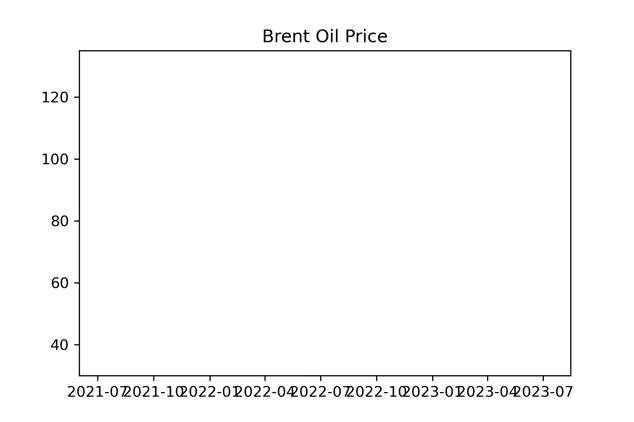
**Animation**

Let’s proceed to animate our line chart. To achieve this, we’ll utilize the FuncAnimation module from Matplotlib. Think of animations with FuncAnimationas crafting a film. Just as movies display a sequence of images, we'll sequentially plot our data points until the entire chart materializes. The approach is analogous: we'll start by setting up a figure. However, we'll hold off on adding the previously mentioned line graph, as it will be incorporated in a subsequent step:

fig, ax = plt.subplots(figsize=(6,4))  
  
# Set title  
ax.set\_title('Brent Oil Price')  
  
# Set the min and max dates  
min\_date = datetime.strptime('2021-06-01', '%Y-%m-%d')  
max\_date = datetime.strptime('2023-08-15', '%Y-%m-%d')  
  
# Set the limits for the axes  
ax.set\_ylim([30, 135])  
ax.set\_xlim([min\_date,max\_date])

To ensure our axes remain static during the animation while data is smoothly added, we’ve set fixed limits for both axes using ax.set\_ylim and ax.set\_xlim. With that in place, let's move to the animation:

# Initialize the lines  
line, = ax.plot([], [], color= '#884b8f', linestyle='-')  
  
def init(): # Initialization function  
 line.set\_data([], [])  
 return line  
  
def animate(i): # Animation function  
 line.set\_data(dates\_vals[:i], brent.iloc[:i])  
 return line  
  
#Animation execution  
ani = FuncAnimation(fig=fig, func=animate, frames=len(brent),   
 init\_func=init, blit=True)



Works just fine!. Let’s break down the code. First, we define our baseline (empty) line plot:

# Initialize the line  
line, = ax.plot([], [], color= '#884b8f', linestyle='-')

Moving forward, we’ll establish our init() and animate() functions using FuncAnimation. The init() function determines the starting point of our animation. For our purposes, it will initiate with a blank canvas, which we'll progressively fill.

def init(): # Initialization function  
 line.set\_data([], [])  
 return line

With the animate() function in place, we'll sequentially introduce data using the set\_data() method. This method allows us to refresh and display individual data points in succession. For every frame (i), we'll plot the corresponding value (i) and its associated index (i) - which, in this context, represents a date.

def animate(i): # Animation function  
 line.set\_data(dates\_vals[:i], brent.iloc[:i])  
 return line

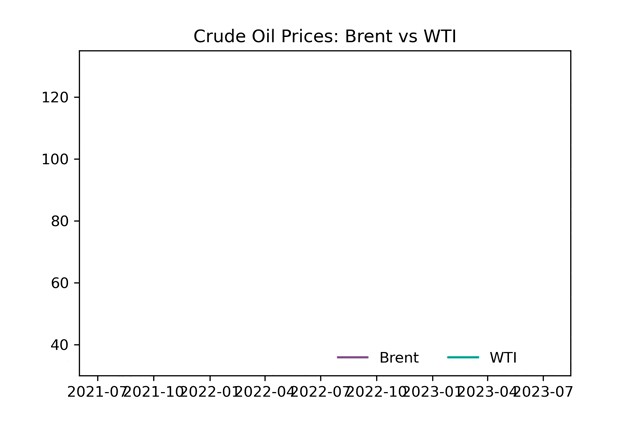
Now, let’s dive into the core lines where the true magic happens. We’ll invoke FuncAnimation(), utilizing our pre-established figure (fig), animation function (animate(i)), and initialization function (init(i)). We've designated frames=len(brent) to ensure the animation encompasses all time points from our 'brent' dataframe. Lastly, by setting the 'blit' parameter to True, we ensure that only new objects are rendered in the animation, significantly enhancing its drawing efficiency.

#Animation  
ani = FuncAnimation(fig=fig, func=animate, frames=len(brent),   
 init\_func=init, blit=True)

**What if we wanted to add more data?**

The procedure is more or less the same. We’ll initiate a second empty line and incorporate it into both the init() and animate() functions. This ensures both data lines are concurrently updated and visualized in the animation. Let's integrate the WTI crude oil data into our animation as demonstrated below:

# Initialize the lines  
line, = ax.plot([], [], color= '#884b8f', linestyle='-', label='Brent')  
line1, = ax.plot([], [], color= '#00A88F', linestyle='-', label='WTI')  
  
def init(): # Initialization function  
 line.set\_data([], [])  
 line1.set\_data([], [])  
 return line  
  
def animate(i):  
 line.set\_data(dates\_vals[:i], brent.iloc[:i])  
 line1.set\_data(dates\_vals1[:i], wti.iloc[:i])  
 return line  
  
#Animation  
ani = FuncAnimation(fig=fig, func=animate, frames=len(brent),   
 init\_func=init, blit=True)



**Visual customization**

We’ve got our animated line charts up and running. Now, let’s make them look even better. The cool thing about ***Matplotlib***is that you can change just about anything to make your chart look the way you want. If you’re curious about all the ways you can tweak things, check out the [*Matplotlib*](https://matplotlib.org/stable/users/index.html)full guides and tutorials.

Below, I’ve shared some of the changes we made and the result:

**Title:**

ax.set\_title('Crude Oil: Spot Prices (USD/bl)', pad=20, fontsize=20,   
 font='roboto', weight='bold')

**Subtitle:**

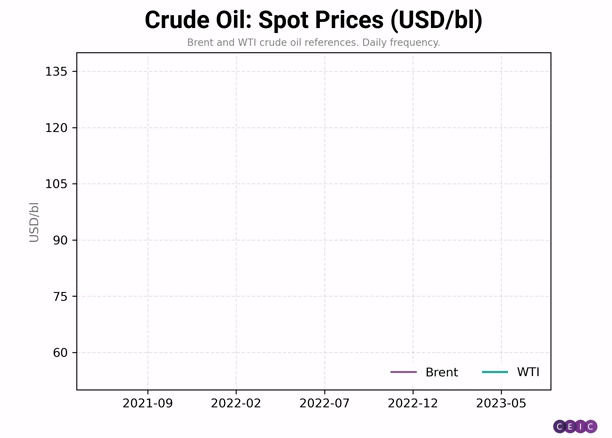
ax.annotate('Brent and WTI crude oil references. Daily frequency.',  
 xy=(0.5,1.02), xytext=(0,0), xycoords='axes fraction',  
 textcoords='offset points', ha='center', fontsize=8, color='grey')

**Grid:**

ax.grid(color='grey', alpha=0.2, linestyle='--')

**Date axis format:**

ax.xaxis.set\_major\_locator(ticker.MaxNLocator(nbins=8))  
ax.xaxis.set\_major\_formatter(DateFormatter('%Y-%m'))



Much better! right?

**Bonus: Annotations**

Charts don’t always speak for themselves. Sometimes, we need to add notes or comments to highlight specific points or ideas. Just like before, with Matplotlib, we can add pictures, text, or pretty much anything else we want. If you’re looking to add static notes you can use the ax.annotation() method from our earlier chart. But if you want notes to show up at a certain date or data point, you’ll need to add them into the animation functions.

First, we are going to create a dictionary with the annotations and then create a list from them. Note also that we are including an arrow and its properties to the visualization:

#Annotations  
annotations\_info = [  
 {  
 'date': datetime.strptime('2022-03-10', '%Y-%m-%d'),  
 'props': {  
 'text': 'Prices rally on \nuncertainty over \nRussia-Ukraine \nconflict',  
 'xy': (0.35, 0.85),  
 'xytext': (-70, 0),  
 'xycoords': 'axes fraction',  
 'textcoords': 'offset points',  
 'fontsize': 8,  
 'ha': 'center',  
 'va': 'baseline',  
 'color': 'black',  
 'weight': 'normal',  
 'arrowprops': {  
 'arrowstyle': '->',  
 'color': 'grey'  
 }  
 },  
 'visible': False, # This flag will indicate whether the annotation should be visible  
 },  
 {  
 'date': datetime.strptime('2022-12-15', '%Y-%m-%d'),  
 'props': {  
 'text': 'Prices fall amid \nweaker demand \nfrom China',  
 'xy': (0.7, 0.2),  
 'xytext': (-100, 0),  
 'xycoords': 'axes fraction',  
 'textcoords': 'offset points',  
 'fontsize': 8,  
 'ha': 'left',  
 'va': 'top',  
 'color': 'black',  
 'weight': 'normal',  
 'arrowprops': {  
 'arrowstyle': '->',  
 'color': 'grey'  
 }  
 },  
 'visible': False, # This flag will indicate whether the annotation should be visible  
 }  
]  
  
# Create the annotations and store them in a list  
annotations = [ax.annotate(\*\*info['props'], visible=info['visible']) for info in annotations\_info]

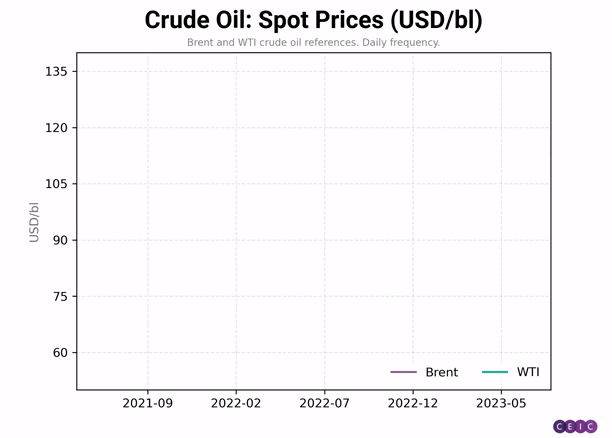
Here we set the parameter ‘*visible*’ to False to ensure that the annotation is not displayed from the beginning. In contrast, we will use the *‘date’* parameter to define that the annotation will be displayed when the index of the data reaches that specific date.

Now we need to include the annotations in our initialization function so that the program knows that we are going to include them in the animation

def init(): # Initialization function  
 line.set\_data([], [])  
 line1.set\_data([], [])  
  
 for annotation in annotations:  
 annotation.set\_visible(False)  
   
 return line, line1, \*annotations

In the animate(i) function, we'll loop through our annotations list. Each annotation will show up when the index matches the date we've set in the 'date' parameter. We'll then set the set\_parameter to True to make sure each annotation appears correctly.

def animate(i):  
 line.set\_data(dates\_vals[:i], brent.iloc[:i])  
 line1.set\_data(dates\_vals1[:i], wti.iloc[:i])  
   
 # Update the visibility of annotations  
 current\_date = dates\_vals[i].date()  
 for annotation, info in zip(annotations, annotations\_info):  
 if current\_date >= info['date'].date() and not annotation.get\_visible():  
 annotation.set\_visible(True)  
   
 return line, line1, \*annotations



And that’s it! Do you realise how the notes give more context about the chart?

**Exporting/saving**

We can save our animation in GIF or MP4 format. The first thing to do is to install the FFMpeg or Pillow modules in order to render the animations. Now we have to define two key parameters.

The first one is the *fps (*frames per second), which will give us the drawing speed of our data points, the higher we set the *fps*parameter the faster the information will be displayed. Second, we will define our *dpi (*dots per inch) parameter, which will determine the quality of the image, the higher the *dpi*the better resolution we will get in the end. Note that the higher the *dpi*the longer it takes for the program to render the image.

# Save as mp4  
f = r"oil\_prices\_final\_annotations.mp4"  
writervideo = animation.FFMpegWriter(fps=30)   
ani.save(f, writer=writervideo, dpi=300)  
  
#Save as gif  
f = r"basic\_anim\_init.gif"   
writergif = animation.PillowWriter(fps=300)   
ani.save(f, writer=writergif, dpi=300)

If you have any additional questions or are interested in having a sample of the data we use for the exercise, please do not hesitate to contact us through our [website](https://www.ceicdata.com/en)or [social media](https://www.linkedin.com/company/ceic-data/).