Lecture 3: Time Series

Temporal data analysis

A *time series* is a series of data points indexed by time, $x_i = x(t_i)$, for $i = 1, \ldots, n$.
Examples frequently occur in

- weather forecasting,
- mathematical finance (stocks),
- electricity demand in a power grid,
- keystrokes on a computer, and
- any applied science and engineering which involves temporal measurements

Temporal data analysis or *time series analysis* is just the study of such data.

As a first example of time series data, we'll consider stocks and *mathematical finance*.

Mathematical finance

- Advanced mathematics, such as analysis of the Black-Scholes model, is now essential to finance.
- Algorithms are now responsible for making split-second decisions. In fact, the speed at which light travels is a limitation when designing trading systems.
- Machine learning and data mining techniques are popular in the financial sector. For example, **high-frequency trading (HFT)** is a branch of algorithmic trading where computers make thousands of trades in short periods of time, engaging in complex strategies such as statistical arbitrage and market making. HFT was responsible for phenomena such as the 2010 flash crash and a 2013 flash crash prompted by a hacked Associated Press tweet about an attack on the White House.

The `pandas_datareader` package

Functions from `pandas_datareader.data` and `pandas_datareader.wb` extract data from various internet sources into a pandas DataFrame.

We will use the function

```
df = pandas_datareader.data.DataReader(name, data_source=None, start=None, end=None,
      retry_count=3, pause=0.1, session=None, api_key=None)
```

to import stock data as a pandas DataFrame. The arguments that we'll use are

```
name : str or list of strs
    the name of the dataset. Some data sources (IEX, fred) will accept a list of names.
data_source: {str, None}
    the data source ("iex", "fred", "ff")
start : string, int, date, datetime, Timestamp
    left boundary for range (defaults to 1/1/2010)
end : string, int, date, datetime, Timestamp
    right boundary for range (defaults to today)
```

Getting and Visualizing Stock Data

```
start = datetime(2011, 1, 1)
end = datetime(2022, 3, 22)

TSLA = web.DataReader(name="TSLA", data_source="yahoo", start=start, end=end)
TSLA.tail()
```

Date	High	Low	Open	Close	Volume	Adj Close
2022-03-17	875.000000	825.719971	830.98999	871.599976	22194300.0	871.599976
2022-03-18	907.849976	867.390015	874.48999	905.390015	33408500.0	905.390015
2022-03-21	942.849976	907.090027	914.97998	921.159973	27327200.0	921.159973
2022-03-22	997.859985	921.750000	930.00000	993.979980	35200400.0	993.979980
2022-03-22	997.799927	921.750000	930.00000	993.979980	35289519.0	993.979980

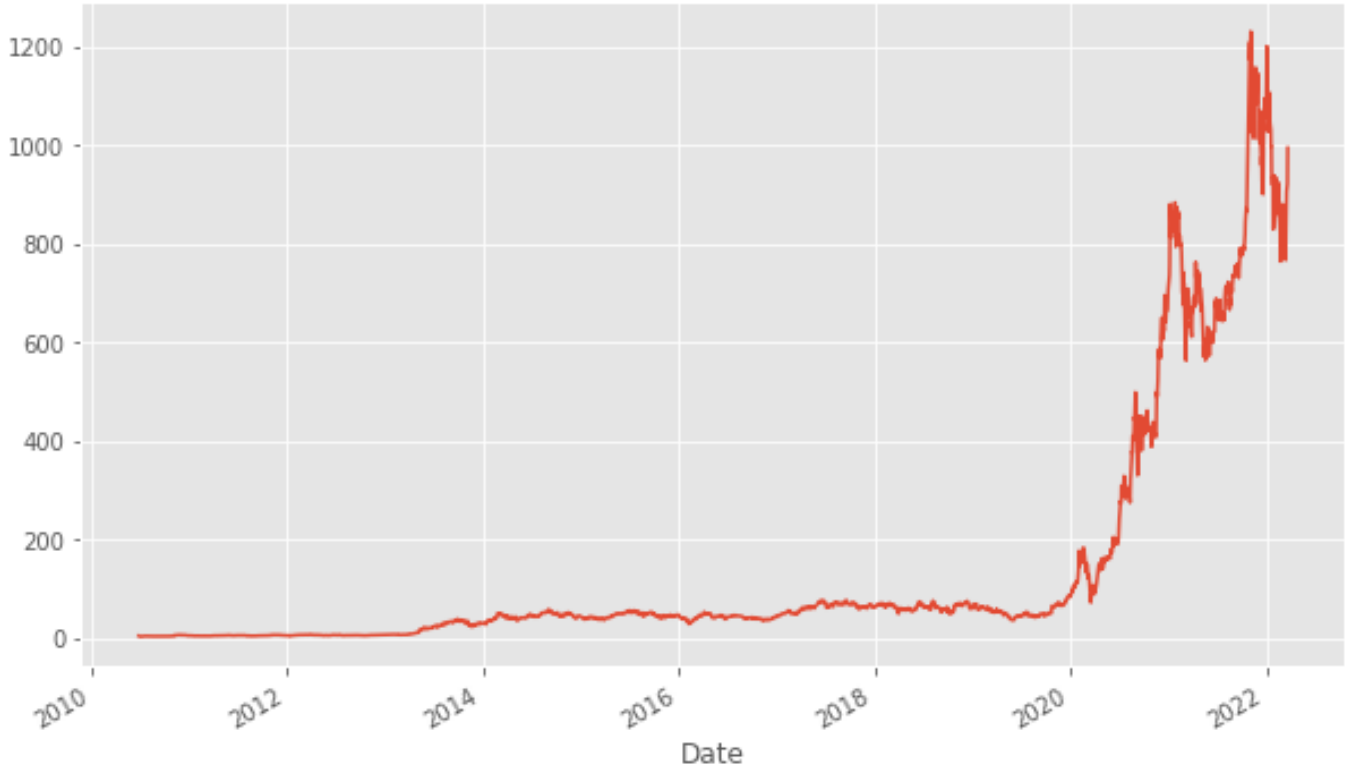
What does this data mean?

- **High** is the highest price of the stock on that trading day,
- **Low** the lowest price of the stock on that trading day,
- **Open** is the price of the stock at the beginning of the trading day (it need not be the closing price of the previous trading day)
- **Close** the price of the stock at closing time
- **Volume** indicates how many stocks were traded
- **Adj Closed** is the price of the stock after adjusting for corporate actions. While stock prices are considered to be set mostly by traders, *stock splits* (when the company makes each extant stock worth two and halves the price) and *dividends* (payout of company profits per share) also affect the price of a stock and should be accounted for.

Visualizing Stock Data

Now that we have stock data we can visualize it using the `matplotlib` package, called using a convenience method, `plot()` in pandas.

```
TSLA["Adj Close"].plot(grid = True); # Plot the adjusted closing price of TSLA
```



Plotting multiple stocks together

For a variety of reasons, we may wish to plot multiple financial instruments together including:

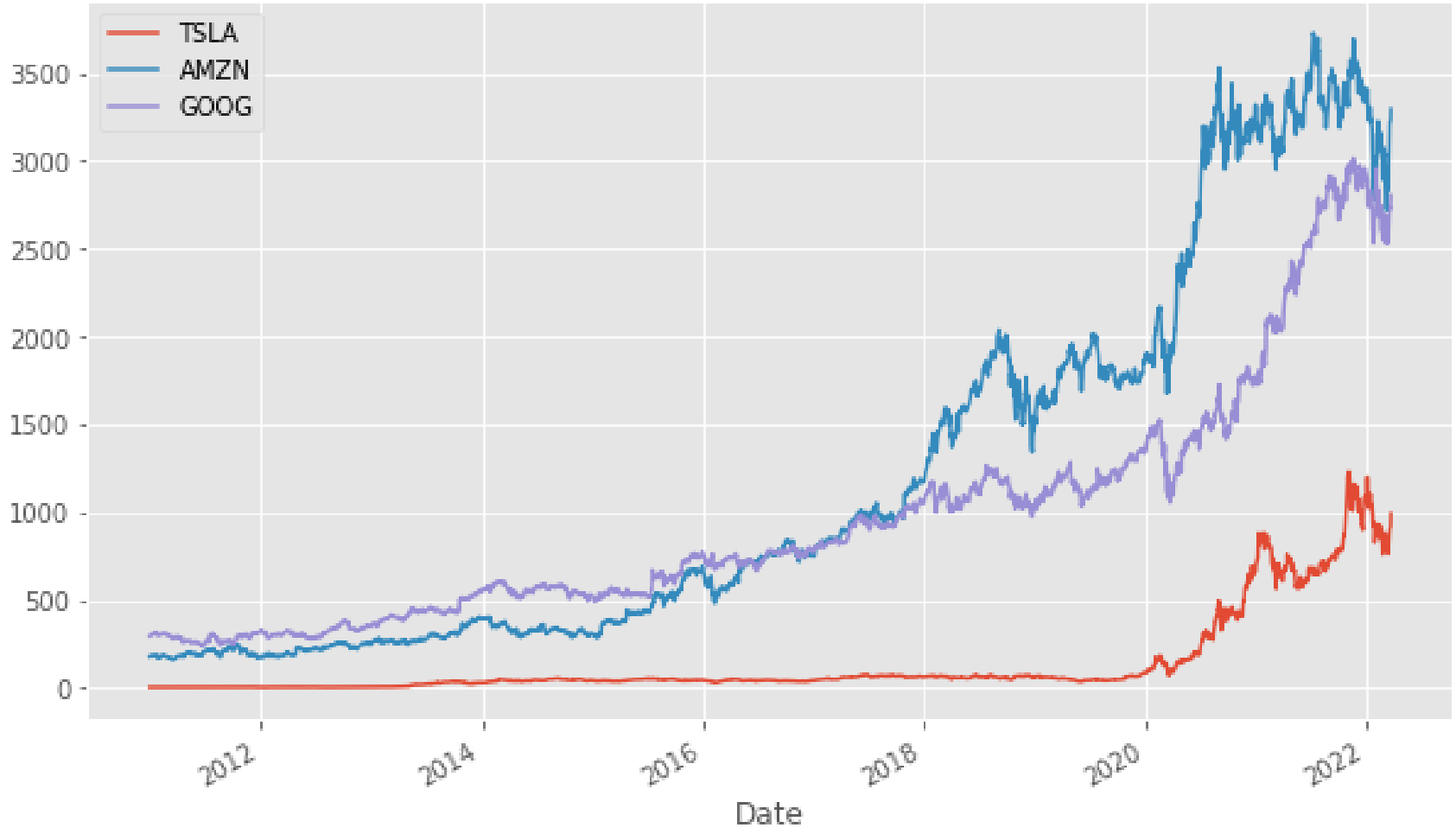
- we may want to compare stocks
- compare them to the market or other securities such as exchange-traded funds (ETFs).

Here, we plot the adjusted close for several stocks together.

```
AMZN, GOOG = (web.DataReader(name=s, data_source="yahoo", start=start, end=end) for s in ["AMZN", "GOOG"])

adj_close = pd.DataFrame({ "TSLA": TSLA["Adj Close"],
                           "AMZN": AMZN["Adj Close"],
                           "GOOG": GOOG["Adj Close"]})

adj_close.head()
```

Date	TSLA	AMZN	GOOG
2010-12-31	5.326	180.000000	295.875977
2011-01-03	5.324	184.220001	301.046600
2011-01-04	5.334	185.009995	299.935760
2011-01-05	5.366	187.419998	303.397797
2011-01-06	5.576	185.860001	305.604523

```
adj_close.plot(grid = True)
```

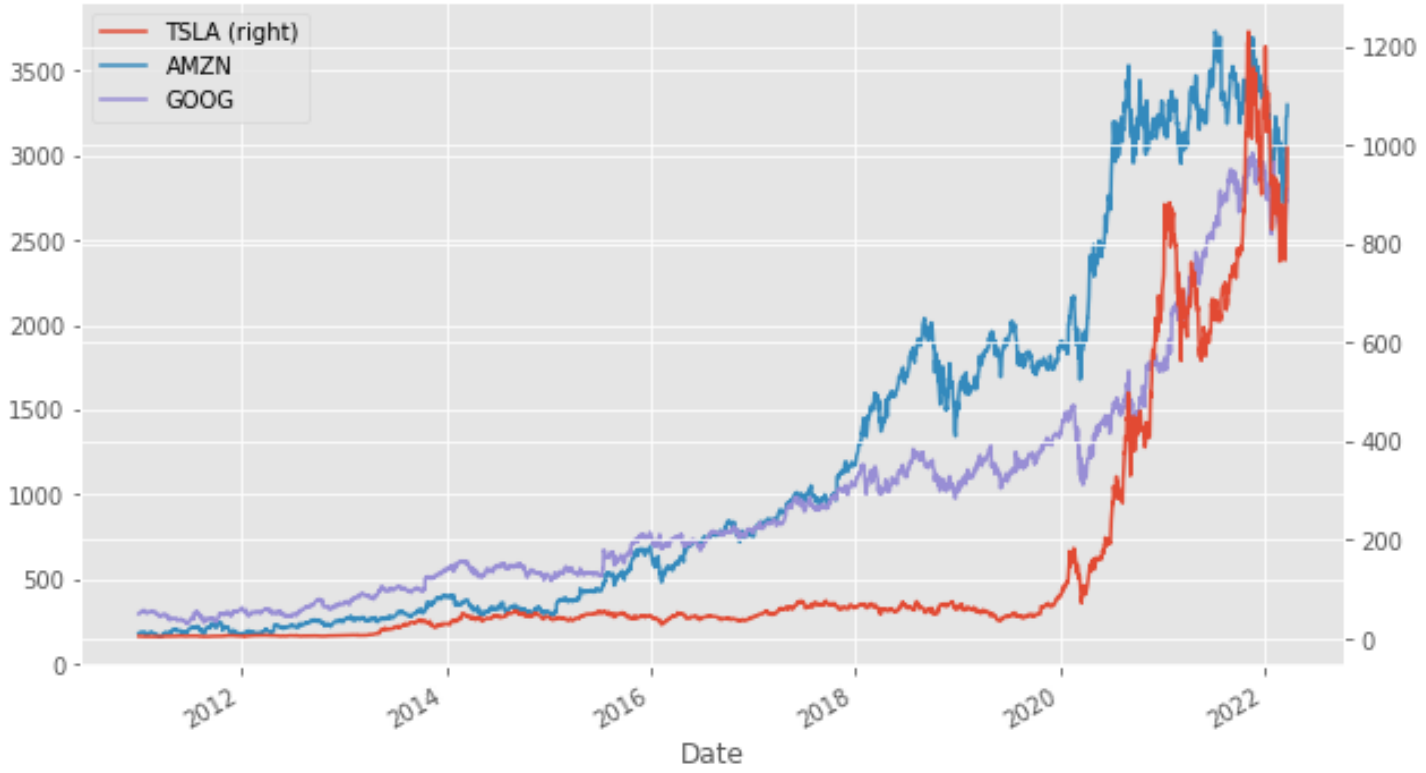


Q: Why is this plot difficult to read?

It plots the *absolute price* of stocks with time. While absolute price is important, frequently we are more concerned about the *relative change* of an asset rather than its absolute price. Also, Amazon and Google stock is much more expensive than Tesla stock, and this difference makes Tesla stock appear less volatile than they truly are (that is, their price appears not to vary as much with time).

One solution is to use two different scales when plotting the data; one scale will be used by Amazon and Google stocks, and the other by Tesla.

```
adj_close.plot(secondary_y = ["TSLA"], grid = True)
```



But, this solution clearly has limitations. We only have two sides of the plot to add more labels!

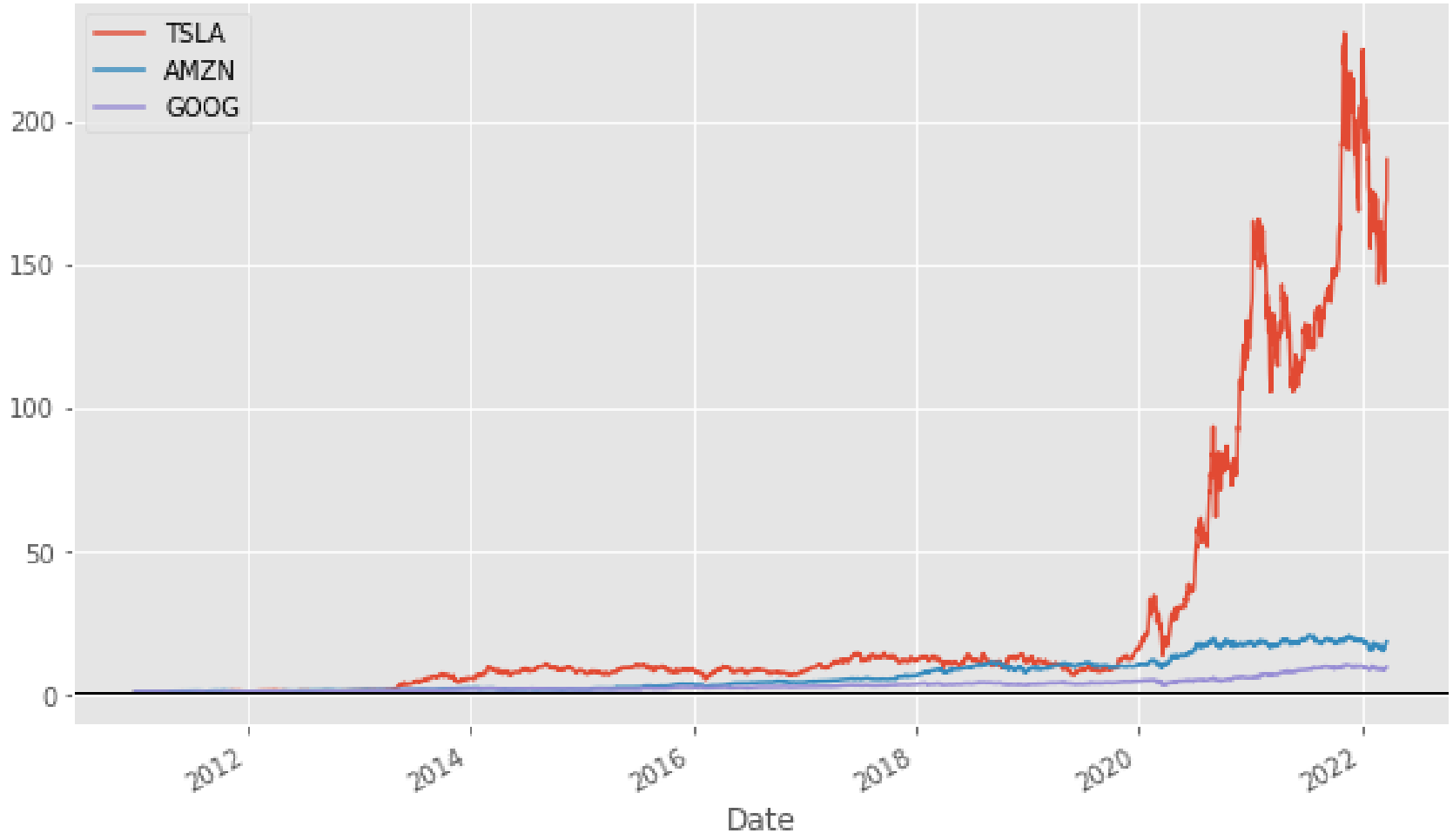
A "better" solution is to plot the information we actually want. One option is to plot the *stock returns since the beginning of the period of interest*:

$$\text{return}_{t,0} = \frac{\text{price}_t}{\text{price}_0}$$

This requires transforming the data, which we do using a *lambda function*.

```
stock_return = adj_close.apply(lambda x: x / x[0])
stock_return.head()
```

```
stock_return.plot(grid = True).axhline(y = 1, color = "black", lw = 1);
```



This is a much more useful plot!:

- We can now see how profitable each stock was since the beginning of the period.
- Furthermore, we see that these stocks are highly correlated; they generally move in the same direction, a fact that was difficult to see in the other charts.

Alternatively, we could plot the change of each stock per day. One way to do so would be to use the *percentage increase of a stock*:

$$\text{increase}_t = \frac{\text{price}_t - \text{price}_{t-1}}{\text{price}_t}$$

or the *log difference*.

$$\text{change}_t = \log\left(\frac{\text{price}_t}{\text{price}_{t-1}}\right) = \log(\text{price}_t) - \log(\text{price}_{t-1})$$

Here, $\log$ is the natural log.

Log difference has a desirable property: the sum of the log differences can be interpreted as the total change (as a percentage) over the period summed. Log differences also more cleanly correspond to how stock prices are modeled in continuous time.

We can obtain and plot the log differences of the data as follows.

```
# shift moves dates back by 1.
stock_change = adj_close.apply(lambda x: np.log(x) - np.log(x.shift(1)))
stock_change.plot(grid = True).axhline(y = 0, color = "black", lw = 1);
```

Do you prefer to plot stock return or log difference?

- Looking at returns since the beginning of the period make the overall trend of the securities apparent.
- Log difference, however, emphasizes changes between days.

Comparing stocks to the overall market

We often want to compare the performance of stocks to the performance of the overall market.

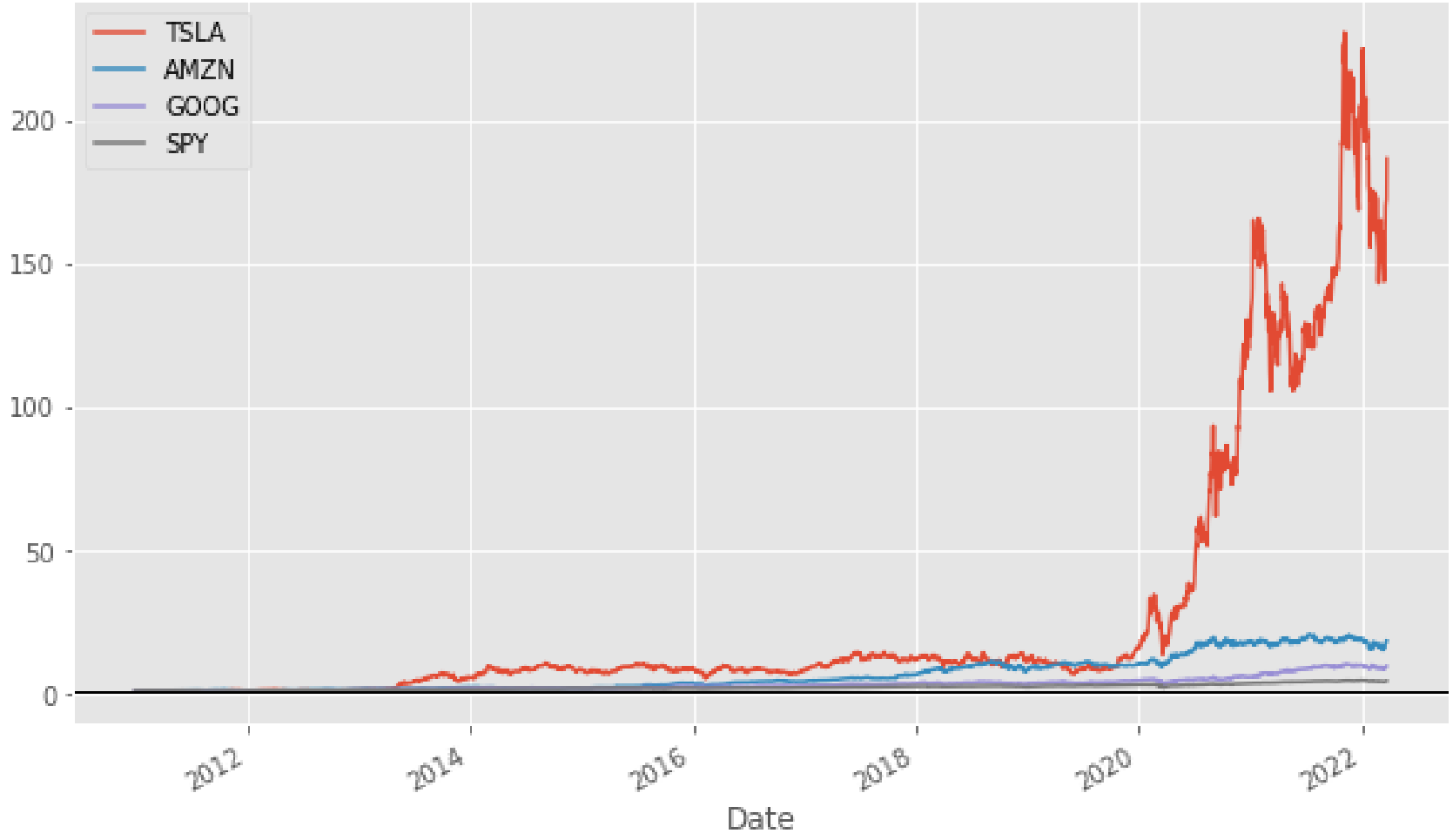
SPY is the ticker symbol for the SPDR S&P 500 exchange-traded mutual fund (ETF), which is a fund that has roughly the stocks in the S&P 500 stock index.

This serves as one measure for the overal market.

```
SPY = web.DataReader(name="SPY", data_source="yahoo", start=start, end=end)
SPY.tail()
```

Date	High	Low	Open	Close	Volume	Adj Close
2022-03-17	441.070007	433.190002	433.589996	441.070007	102676900.0	439.704010
2022-03-18	444.859985	437.220001	438.000000	444.519989	106250400.0	444.519989
2022-03-21	446.459991	440.679993	444.339996	444.390015	88349800.0	444.390015
2022-03-22	450.579987	445.859985	445.859985	449.589996	74539600.0	449.589996
2022-03-22	450.570007	445.880005	445.859985	449.589996	74650394.0	449.589996

```
adj_close['SPY'] = SPY["Adj Close"]
stock_return['SPY'] = adj_close[['SPY']].apply(lambda x: x / x[0])
stock_return.plot(grid = True).axhline(y = 1, color = "black", lw = 1);
```



```
stock_change['SPY'] = adj_close[['SPY']].apply(lambda x: np.log(x) - np.log(x.shift(1)))
stock_change.head()
```

Moving Averages

For a time series x_t, the *q-day moving average at time t*, denoted MA_t^q, is the average of x_t over the past q days,

$$MA_t^q = \frac{1}{q}\sum_{i=0}^{q-1} x_{t-i}$$

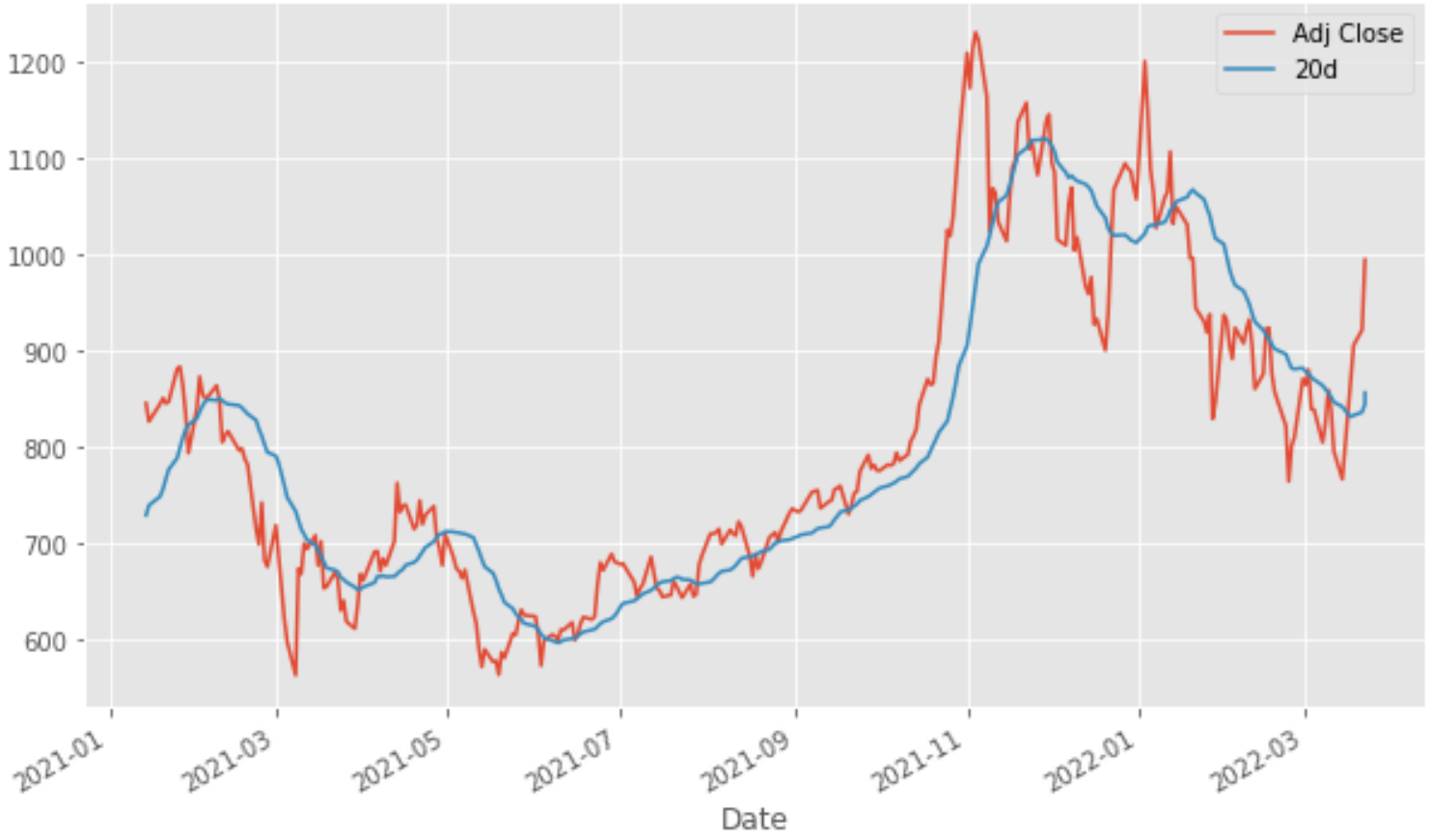
The `rolling` function in Pandas provides functionality for computing moving averages. We'll use it to create a 20-day moving average for Apple stock data and plot it alongside the stock price.

```
TSLA["20d"] = TSLA["Adj Close"].rolling(window = 20, center = False).mean()
TSLA.head(30)
```

Date	High	Low	Open	Close	Volume	Adj Close	20d
2010-12-31	5.450	5.300	5.314	5.326	7089500.0	5.326	NaN
2011-01-03	5.400	5.180	5.368	5.324	6415000.0	5.324	NaN
2011-01-04	5.390	5.204	5.332	5.334	5937000.0	5.334	NaN
2011-01-05	5.380	5.238	5.296	5.366	7233500.0	5.366	NaN
2011-01-06	5.600	5.362	5.366	5.576	10306000.0	5.576	NaN
2011-01-07	5.716	5.580	5.600	5.648	11239500.0	5.648	NaN
2011-01-10	5.736	5.610	5.634	5.690	6713500.0	5.690	NaN
2011-01-11	5.742	5.384	5.718	5.392	8551000.0	5.392	NaN
2011-01-12	5.480	5.304	5.402	5.392	4822000.0	5.392	NaN
2011-01-13	5.394	5.232	5.392	5.244	3618000.0	5.244	NaN
2011-01-14	5.316	5.122	5.230	5.150	5960000.0	5.150	NaN
2011-01-18	5.128	4.950	5.096	5.128	8108500.0	5.128	NaN
2011-01-19	5.094	4.750	5.054	4.806	11857500.0	4.806	NaN
2011-01-20	4.890	4.474	4.806	4.524	11399500.0	4.524	NaN
2011-01-21	4.718	4.542	4.624	4.608	6085000.0	4.608	NaN
2011-01-24	4.962	4.646	4.706	4.898	8225500.0	4.898	NaN
2011-01-25	4.978	4.804	4.930	4.936	6357500.0	4.936	NaN
2011-01-26	4.976	4.820	4.942	4.950	5399500.0	4.950	NaN
2011-01-27	5.016	4.906	4.948	4.984	4478500.0	4.984	NaN
2011-01-28	4.976	4.750	4.976	4.802	5242000.0	4.802	5.1539
2011-01-31	4.824	4.700	4.810	4.820	4151500.0	4.820	5.1286
2011-02-01	4.946	4.708	4.862	4.782	3539000.0	4.782	5.1015
2011-02-02	4.836	4.734	4.832	4.788	2847500.0	4.788	5.0742
2011-02-03	4.780	4.630	4.764	4.726	2560000.0	4.726	5.0422
2011-02-04	4.734	4.644	4.688	4.692	2720000.0	4.692	4.9980

Notice how late the rolling average begins. It cannot be computed until twenty days have passed. Note that this becomes more severe for slower moving averages.

```
TSLA[["Adj Close", "20d"]].tail(300).plot(grid = True);
```

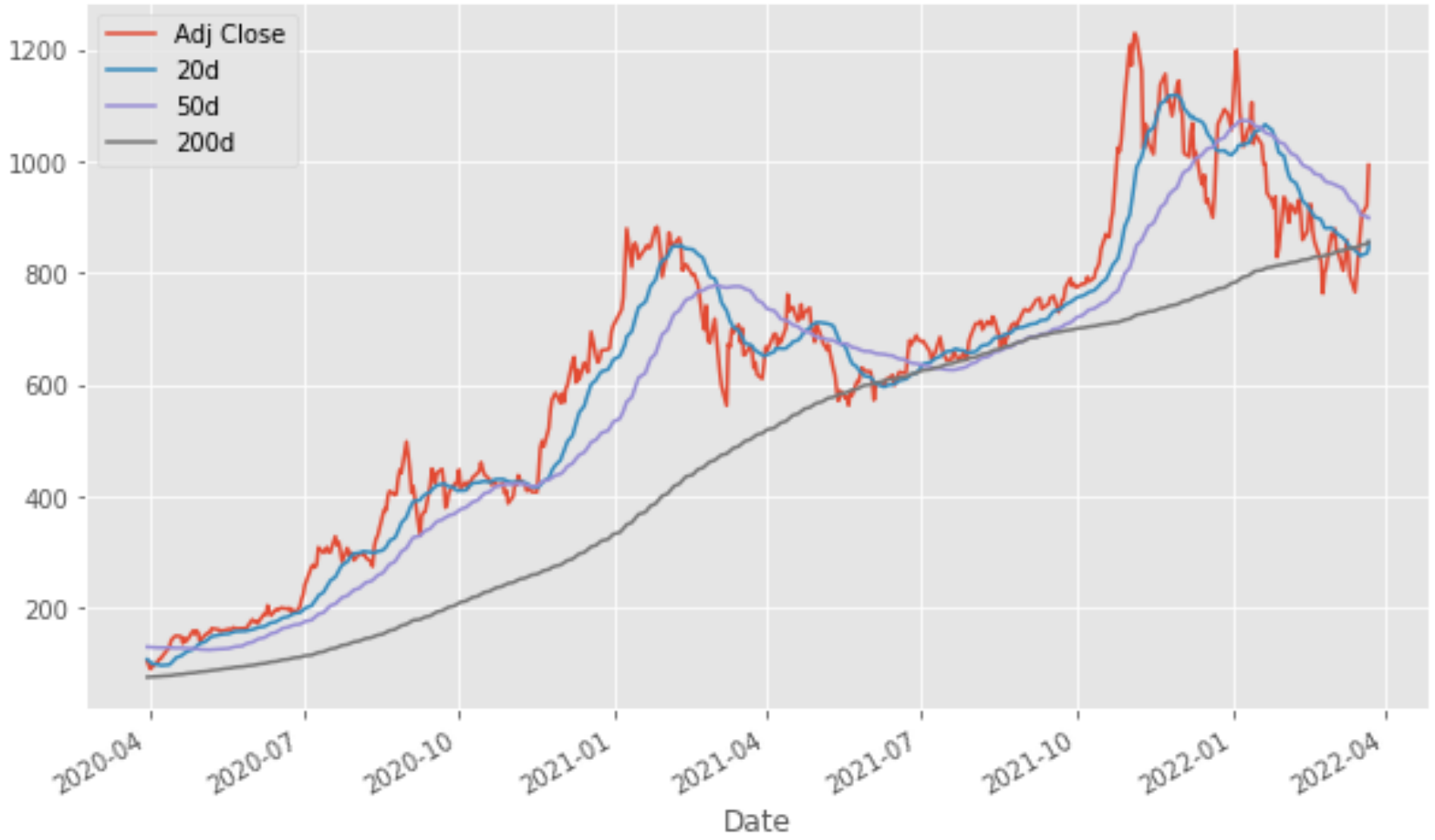


Notice that the moving averages "smooths" the time series. This can sometimes make it easier to identify trends. The larger q, the less responsive a moving average is to fast fluctuations in the series x_t.

So, if these fast fluctuations are considered "noise", a moving average will identify the "signal".

- *Fast moving averages* have smaller q and more closely follow the time series.
- *Slow moving averages* have larger q and respond less to the fluctuations of the stock.

Let's compare the 20-day, 50-day, and 200-day moving averages.



The 20-day moving average is the most sensitive to fluctuations, while the 200-day moving average is the least sensitive.

Trading strategies and backtesting

Trading is the practice of buying and selling financial assets for the purpose of making a profit. Traders develop **trading strategies** that a computer can use to make trades. Sometimes, these can be very complicated, but other times traders make decisions based on finding patterns or trends in charts.

One example is called the moving average crossover strategy.
This strategy is based on two moving averages, a "fast" one and a "slow" one. The strategy is:

- Trade the asset when the fast moving average crosses over the slow moving average.
- Exit the trade when the fast moving average crosses over the slow moving average again.

A trade will be prompted when the fast moving average crosses from below to above the slow moving average, and the trade will be exited when the fast moving average crosses below the slow moving average later.

This is the outline of a complete strategy and we already have the tools to get a computer to automatically implement the strategy.

But before we decide if we want to use it, we should first evaluate the quality of the strategy. The usual means for doing this is called **backtesting**, which is looking at how profitable the strategy is on historical data.

You could now write python code that could implement and backtest a trading strategy. There are also lots of python packages for this:

- **pyfolio** (for analytics)
- **zipline** (for backtesting and algorithmic trading), and
- **backtrader** (also for backtesting and trading).

Time-domain vs frequency-domain analysis

So far, we have thought about a time series $x(t)$ in the "time domain". But, for some time series, it is easier to describe them in terms of the "frequency domain".

For example, a good way to describe the function

$$x(t) = \cos(2\pi f t)$$

is as an oscillating function with frequency f (or period $1/f$).

According to Fourier analysis, we can decompose any signal into its frequency components,

$$x(t) = \sum_{n=-\infty}^{\infty} \hat{x}(n)\, e^{2\pi i n t} \qquad\qquad t \in [0, 1]$$

or

$$x(t) = \int_{-\infty}^{\infty} \hat{x}(f)\, e^{2\pi i f t}\, df \qquad\qquad t \in [-\infty, \infty].$$

The *power spectral density* or *periodogram* $S_{xx}(f) \approx |\hat{x}(f)|^2$ of a time series $x(t)$ describes the distribution of power into the frequency components that compose that signal.

There are lots of time-dependent signals that are periodic or at least some of the signal is periodic. Examples:

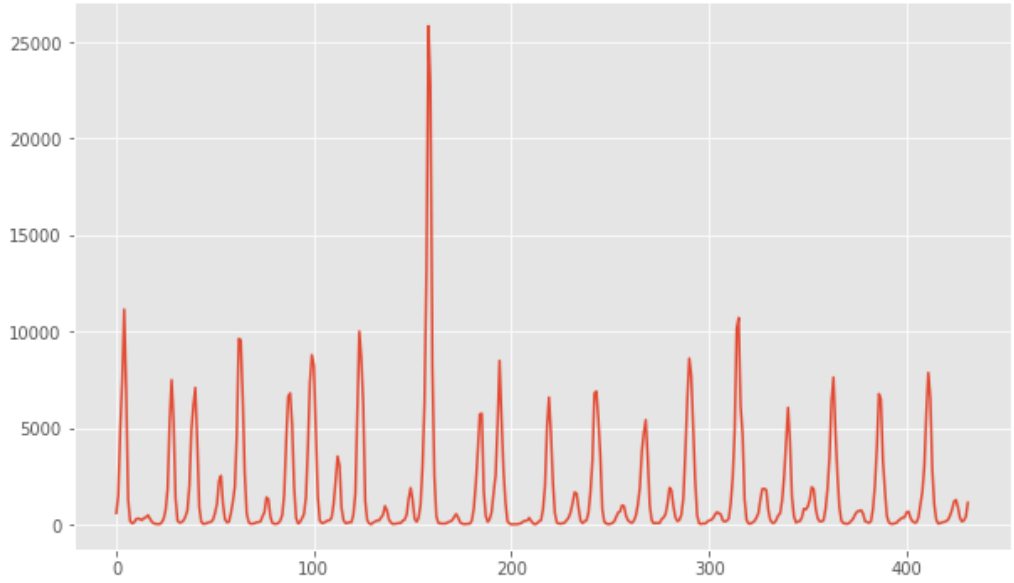
- sunspots follow an 11 year cycle. So if $x(t)$ was a time series representing the "strength" of the sunspot, we would have that $|\hat{x}(f)|^2$ would be large at $f = 1/11$. (Remember period = 1/frequency.)
- The temperature in SLC. Here, we can decompose the temperature into a part that is varying over the course of a year, the part that varies over the day, and the "remaining" part.
- . . .

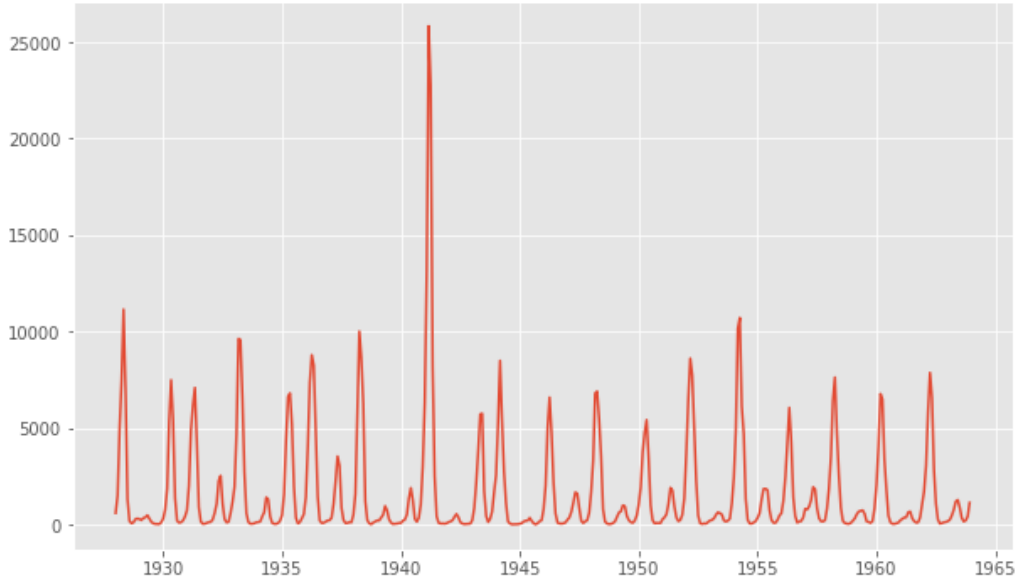
We can compute the power spectral density using the scipy function `periodogram`.

Measles data

We can download the monthly measles data from New York City between 1928 and 1964.

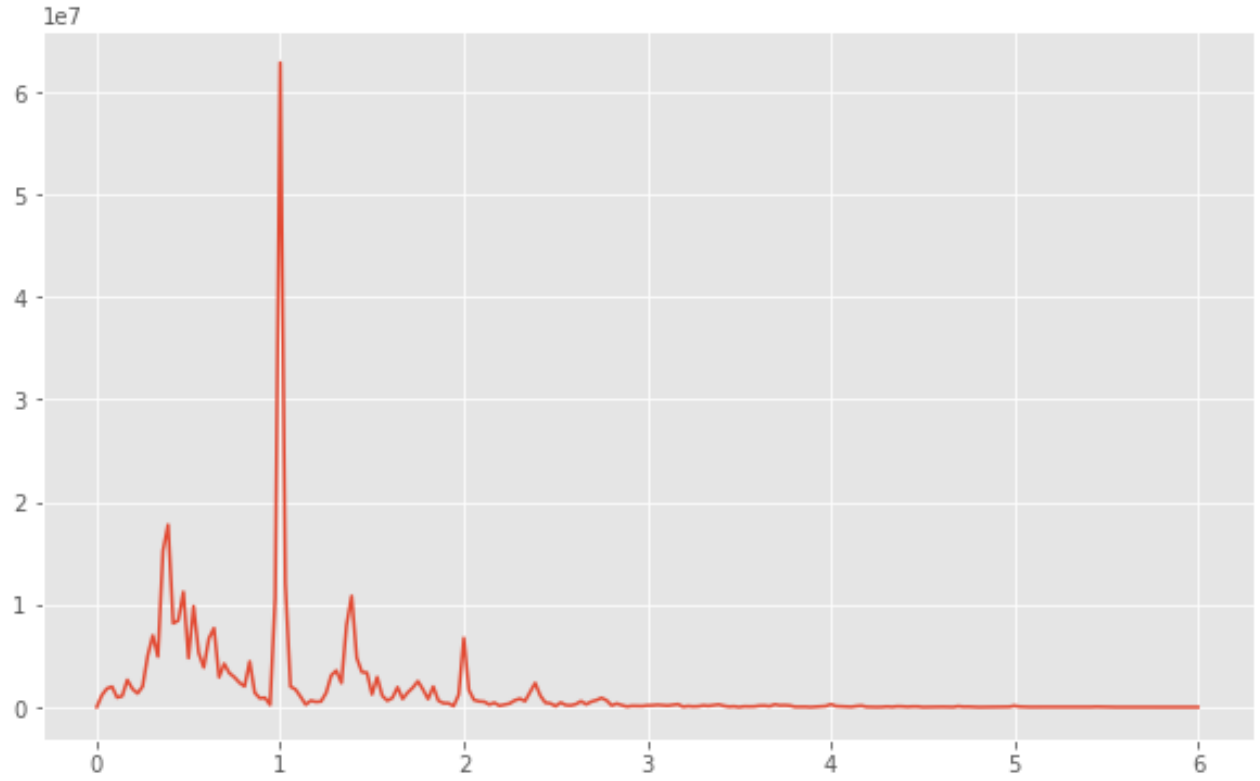
```
df = pd.read_csv("nycmeas.dat", sep=" ", names=["date","cases"])
df["cases"].plot(grid = True);
plt.show()
plt.plot(df["date"].tolist(), df["cases"].tolist())
plt.show()
```





Looking at the plot, we observe that the series is very regular with "periodically occuring" spikes. It appears that approximately once a year, there is a significant measles outbreak. By computing the power spectrum, we can see which frequencies make up this time series.

```
cases = df["cases"].values
f, Pxx_den = signal.periodogram(cases, fs=12, window="hamming")
plt.plot(f, Pxx_den);
```



Since there are 12 months per year, we set the measurement frequency argument in `periodogram` as fs=12.

Clearly, the dominant frequency in this signal is 1 year. Why?

Estimation of COVID-19 Pandemic

Loading Data

We will use data on COVID-19 infected individuals, provided by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University. Dataset is available in this GitHub Repository.

```
base_url = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_time_series/"
infected_dataset_url = base_url + "time_series_covid19_confirmed_global.csv"
recovered_dataset_url = base_url + "time_series_covid19_recovered_global.csv"
deaths_dataset_url = base_url + "time_series_covid19_deaths_global.csv"
countries_dataset_url = base_url + "../UID_ISO_FIPS_LookUp_Table.csv"
```

Let's now load the data for infected individuals and see how the data looks like:

```
infected = pd.read_csv(infected_dataset_url)
infected.head()
```

	Province/State	Country/Region	Lat	Long	1/22/20	1/23/20	1/24/20	1/25/20	1/26/20
0	NaN	Afghanistan	33.93911	67.709953	0	0	0	0	0
1	NaN	Albania	41.15330	20.168300	0	0	0	0	0
2	NaN	Algeria	28.03390	1.659600	0	0	0	0	0
3	NaN	Andorra	42.50630	1.521800	0	0	0	0	0
4	NaN	Angola	-11.20270	17.873900	0	0	0	0	0

5 rows × 795 columns

We can see that each row of the table defines the number of infected individuals for each country and/or province, and columns correspond to dates. Similar tables can be loaded for other data, such as number of recovered and number of deaths.

```
recovered = pd.read_csv(recovered_dataset_url)
deaths = pd.read_csv(deaths_dataset_url)
```

Making Sense of the Data

From the table above the role of province column is not clear. Let's see the different values that are present in `Province/State` column:

```
infected['Province/State'].value_counts()
```

From the names we can deduce that countries like Australia and China have more detailed breakdown by provinces. Let's look for information on China to see the example:

```
infected[infected['Country/Region']=='Australia']
```

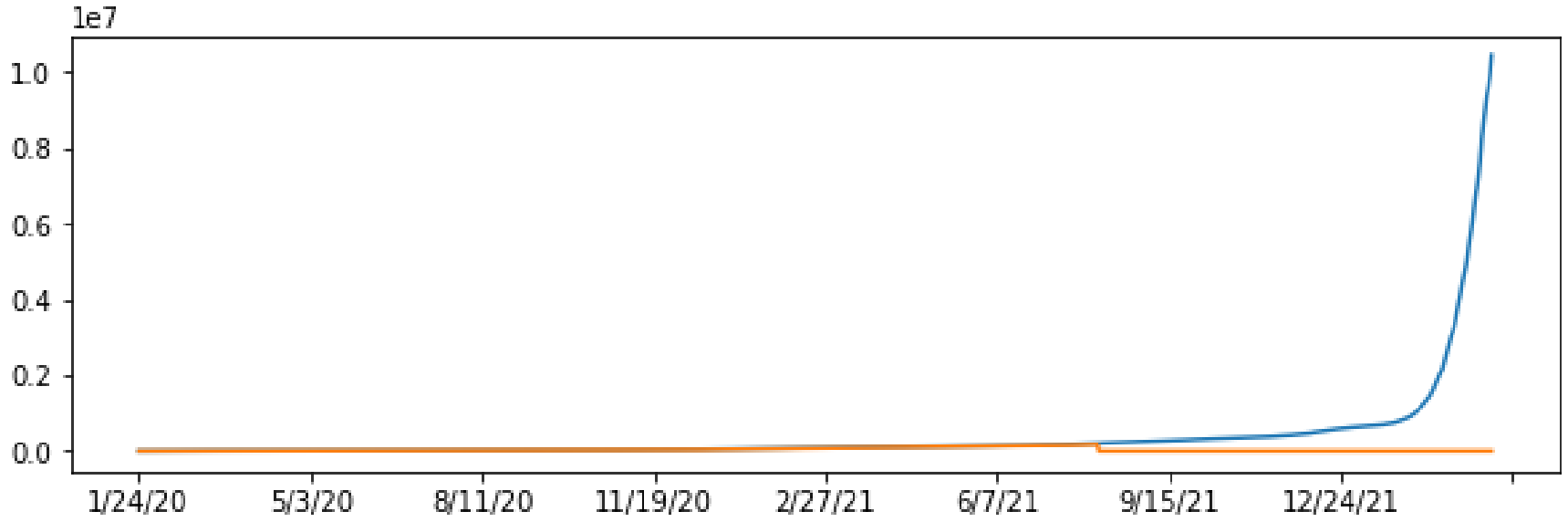
Pre-processing the Data

We are not interested in breaking countries down to further territories, thus we would first get rid of this breakdown and add information on all territories together, to get info for the whole country. This can be done using `groupby`:

```
infected = infected.groupby('Country/Region').sum()
recovered = recovered.groupby('Country/Region').sum()
deaths = deaths.groupby('Country/Region').sum()

infected.head()
```

You can see that due to using `groupby` all DataFrames are now indexed by Country/Region. We can thus access the data for a specific country by using `.loc` :|

```
infected.loc['Korea, South'][2:].plot()
recovered.loc['Korea, South'][2:].plot()
plt.show()
```



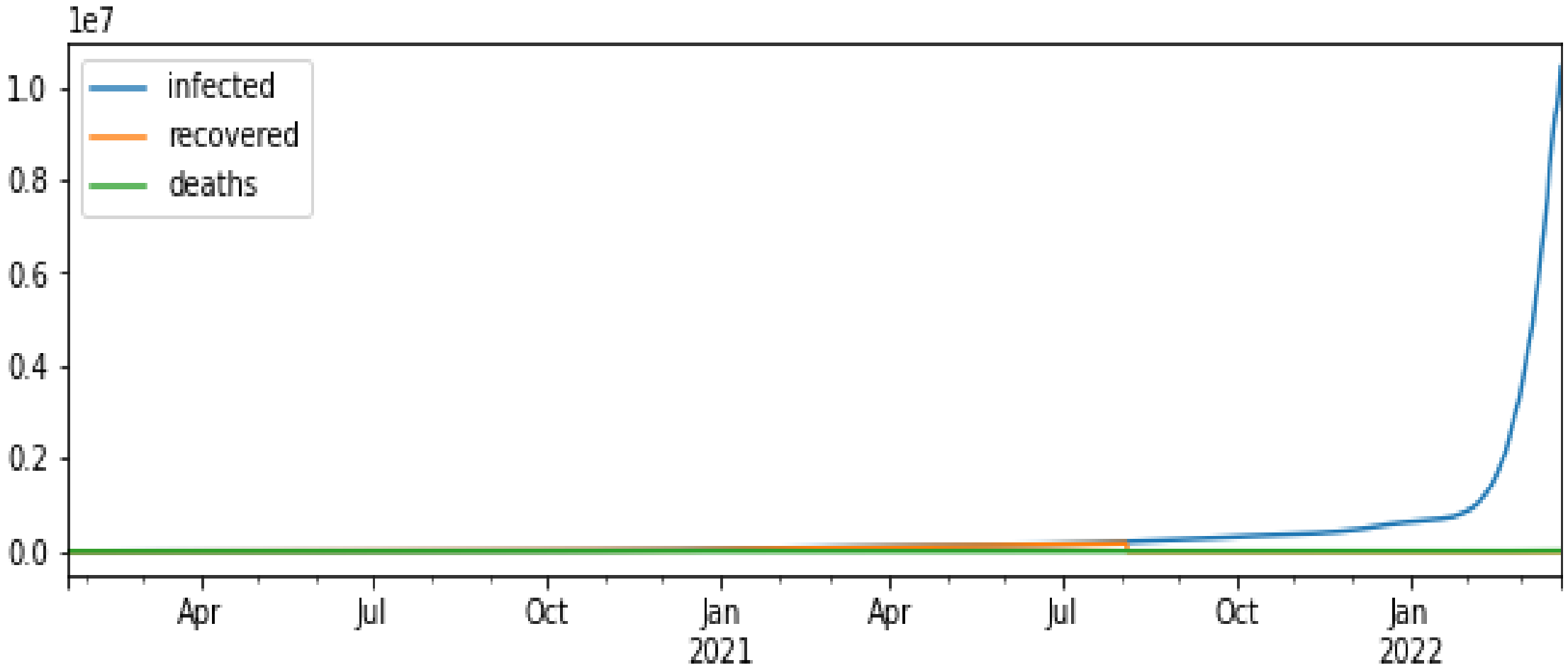
> **Note** how we use `[2:]` to remove first two elements of a sequence that contain geolocation of a country. We can also drop those two columns altogether:

Investigating the Data

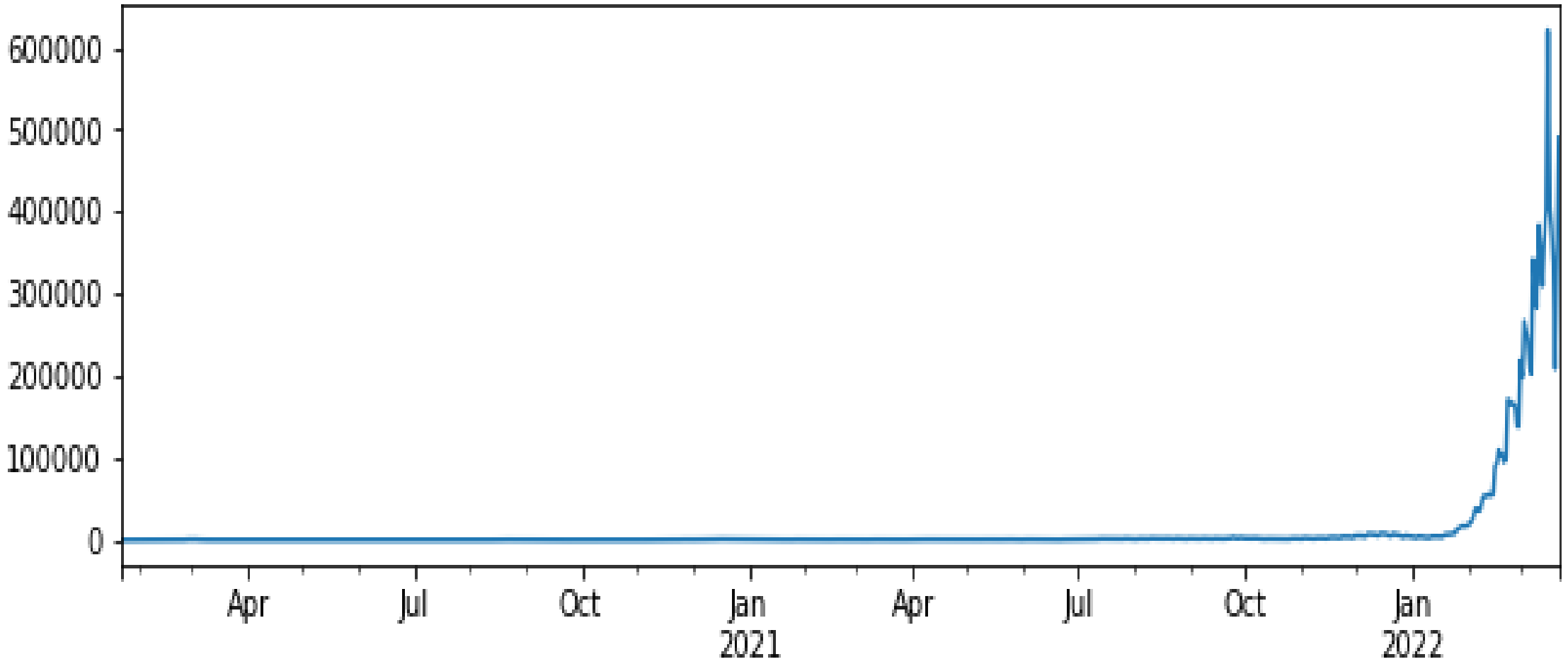
Let's now switch to investigating a specific country. Let's create a frame that contains the data on infections indexed by date:

```
def mkframe(country):
    df = pd.DataFrame({ 'infected' : infected.loc[country] ,
                        'recovered' : recovered.loc[country],
                        'deaths' : deaths.loc[country]})
    df.index = pd.to_datetime(df.index)
    return df

df = mkframe('Korea, South')
df.head()
```

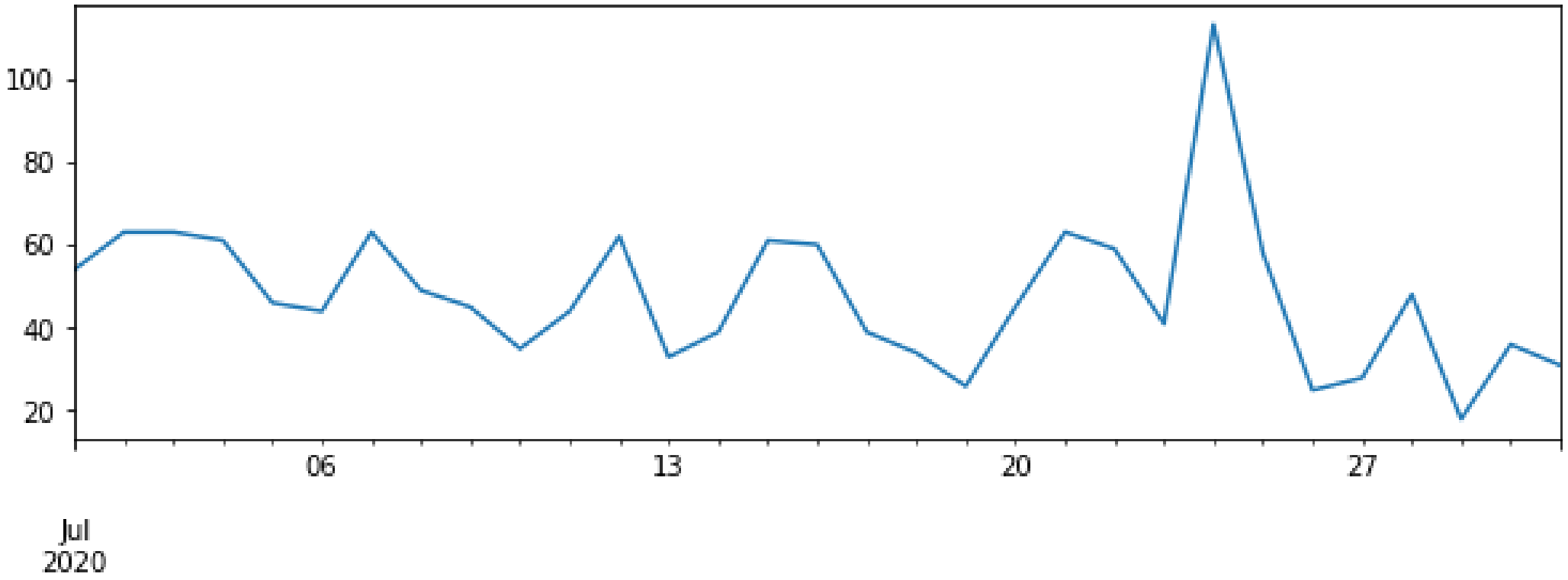
```
df.plot()
plt.show()
```



Now let's compute the number of new infected people each day. This will allow us to see the speed at which pandemic progresses. The easiest day to do it is to use `diff`:

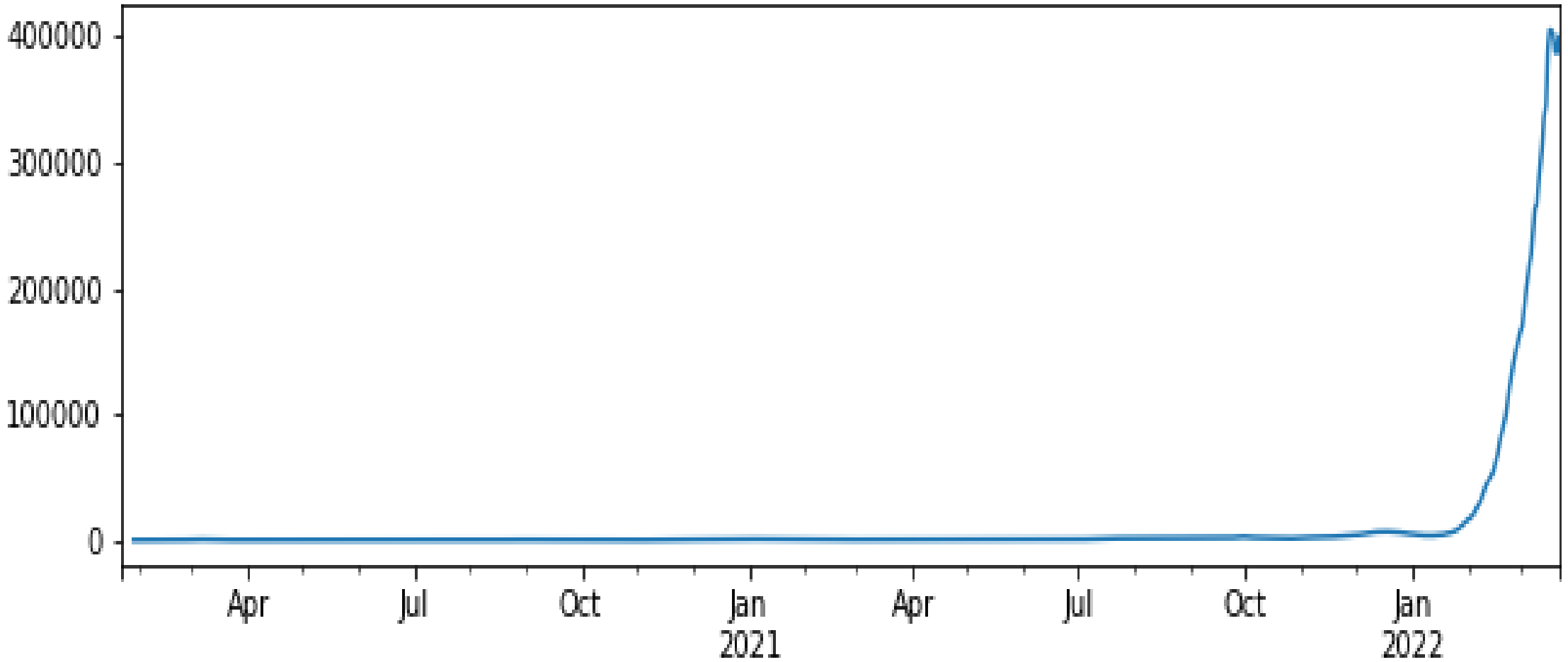


We can see high fluctuations in data. Let's look closer at one of the months:



It clearly looks like there are weekly fluctuations in data. Because we want to be able to see the trends, it makes sense to smooth out the curve by computing running average (i.e. for each day we will compute the average value of the previous several days):

```
df['ninfav'] = df['ninfected'].rolling(window=7).mean()
df['ninfav'].plot()
plt.show()
```



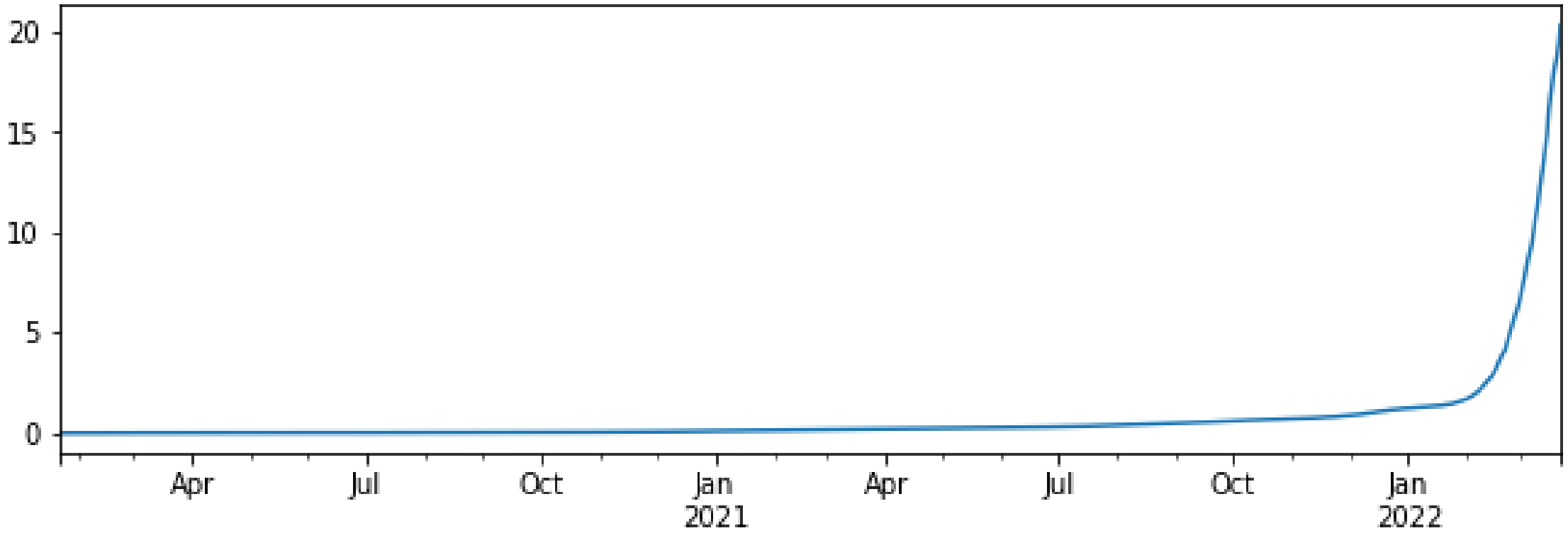
In order to be able to compare several countries, we might want to take the country's population into account, and compare the percentage of infected individuals with respect to country's population. In order to get country's population, let's load the dataset of countries:

```
countries = pd.read_csv(countries_dataset_url)
countries
```

Because this dataset contains information on both countries and provinces, to get the population of the whole country we need to be a little bit clever:

```
countries[(countries['Country_Region']=='Korea, South') & countries['Province_State'].isna()]
```

```
pop = countries[(countries['Country_Region']=='Korea, South') & countries['Province_State'].isna()]['Population'].iloc[0]
df['pinfected'] = df['infected']*100 / pop
df['pinfected'].plot(figsize=(10,3))
plt.show()
```



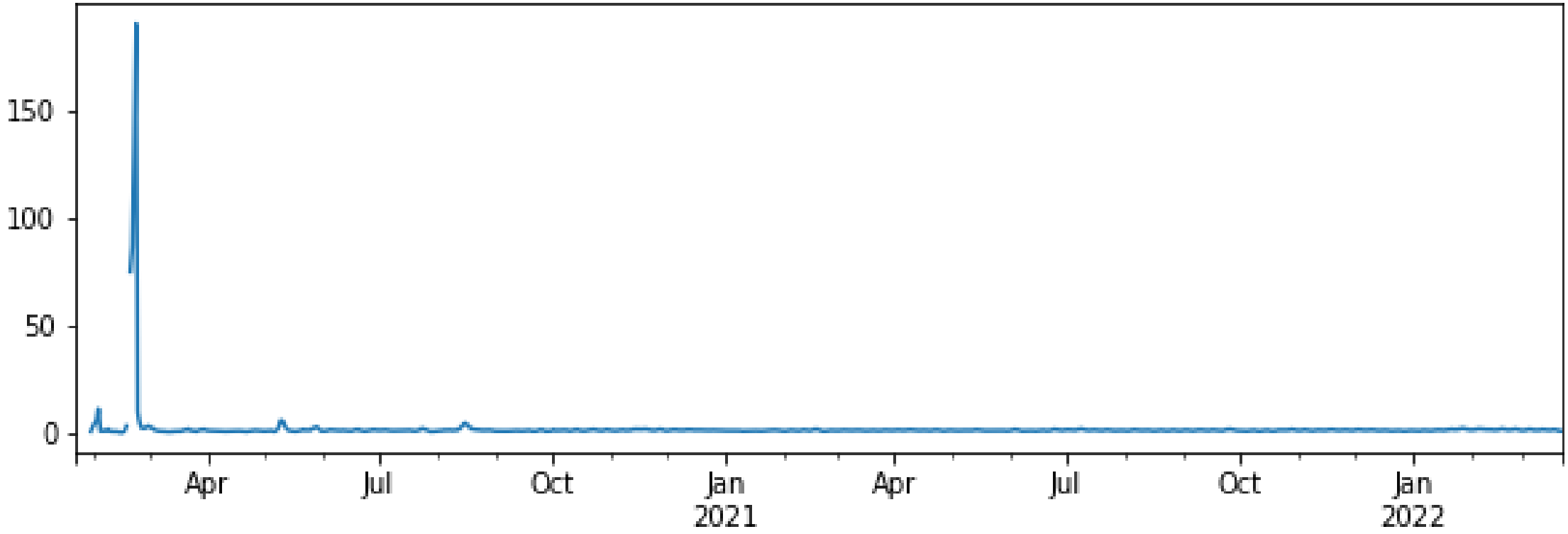
Computing R_t

To see how infectious is the disease, we look at the **basic reproduction number** R_0, which indicated the number of people that an infected person would further infect. When R_0 is more than 1, the epidemic is likely to spread.

R_0 is a property of the disease itself, and does not take into account some protective measures that people may take to slow down the pandemic. During the pandemic progression, we can estimate the reproduction number R_t at any given time t. It has been shown that this number can be roughly estimated by taking a window of 8 days, and computing $$R_t=\frac{I_{t-7}+I_{t-6}+I_{t-5}+I_{t-4}}{I_{t-3}+I_{t-2}+I_{t-1}+I_t}$$ where I_t is the number of newly infected individuals on day t.

Let's compute R_t for our pandemic data. To do this, we will take a rolling window of 8 `ninfected` values, and apply the function to compute the ratio above:

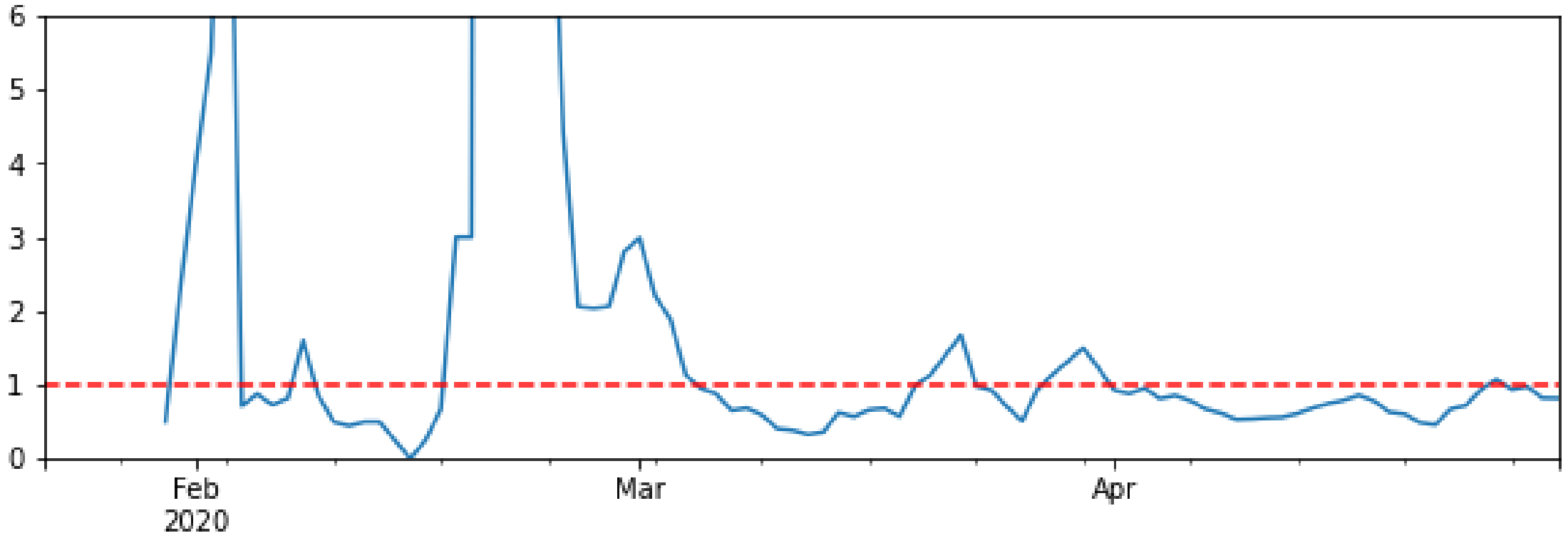
```
df['Rt'] = df['ninfected'].rolling(8).apply(lambda x: x[4:].sum()/x[:4].sum())
df['Rt'].plot()
plt.show()
```



You can see that there are some gaps in the graph. Those can be caused by either `NaN`, if `inf` values being present in the dataset. `inf` may be caused by division by 0, and `NaN` can indicate missing data, or no data available to compute the result (like in the very beginning of our frame, where rolling window of width 8 is not yet available). To make the graph nicer, we need to fill those values using `replace` and `fillna` function.

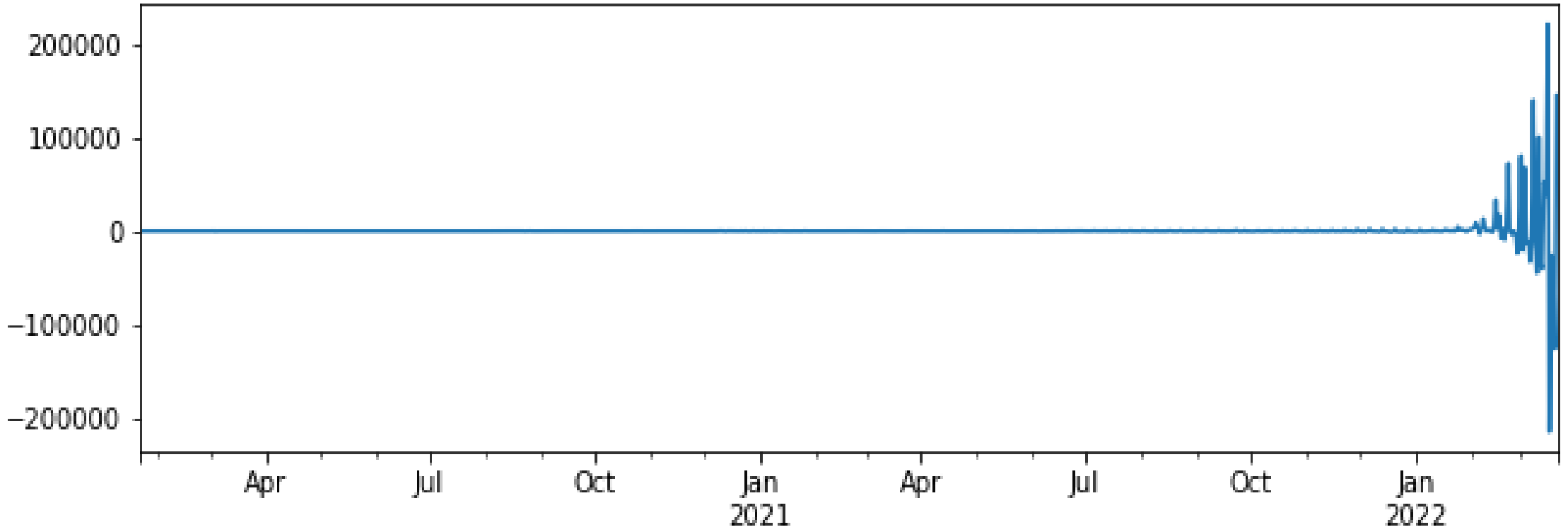
Let's further look at the beginning of the pandemic. We will also limit the y-axis values to show only values below 6, in order to see better, and draw horizontal line at 1.

```
ax = df[df.index<"2020-05-01"]['Rt'].replace(np.inf,np.nan).fillna(method='pad').plot(figsize=(10,3))
ax.set_ylim([0,6])
ax.axhline(1,linestyle='--',color='red')
plt.show()
```



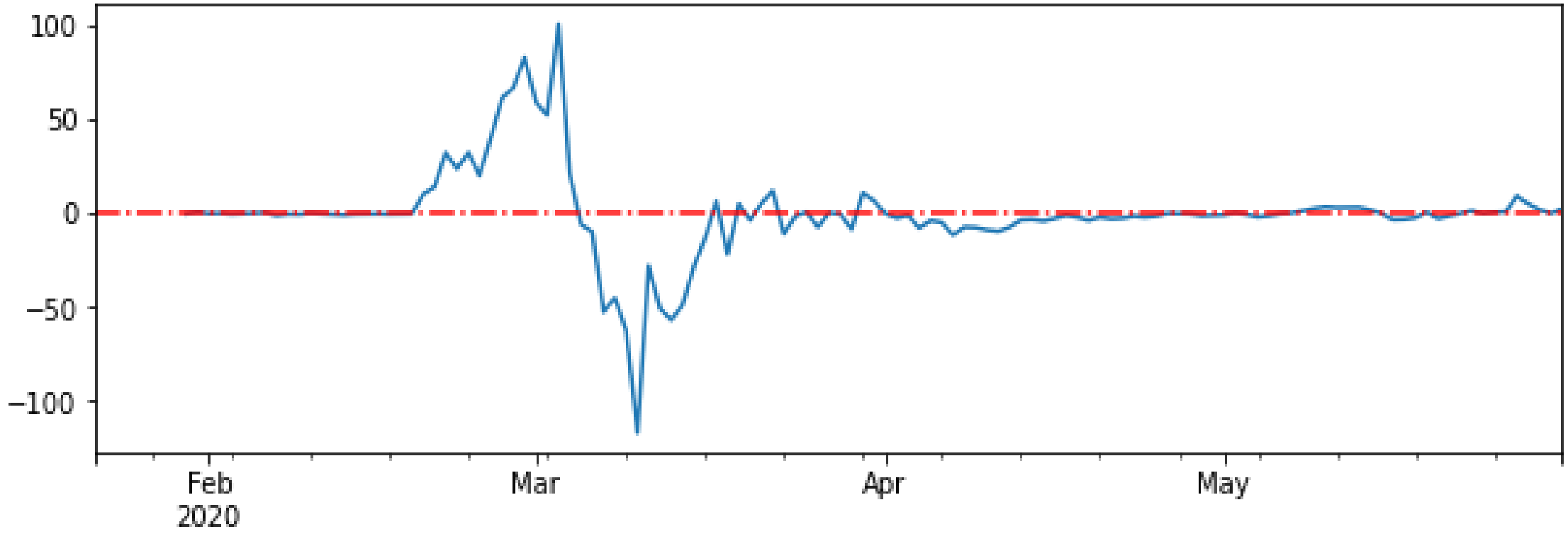
Another interesting indicator of the pandemic is the **derivative**, or **daily difference** in new cases. It allows us to see clearly when pandemic is increasing or declining.

```
df['ninfected'].diff().plot()
plt.show()
```



Given the fact that there are a lot of fluctuations in data caused by reporting, it makes sense to smooth the curve by running rolling average to get the overall picture. Let's again focus on the first months of the pandemic:

```
ax=df[df.index<"2020-06-01"]['ninfected'].diff().rolling(7).mean().plot()
ax.axhline(0,linestyle='-.',color='red')
plt.show()
```



Analyzing COVID-19 Papers

In this challenge, we will continue with the topic of COVID pandemic, and focus on processing scientific papers on the subject. There is CORD-19 Dataset with more than 7000 (at the time of writing) papers on COVID, available with metadata and abstracts (and for about half of them there is also full text provided).

A full example of analyzing this dataset using Text Analytics for Health cognitive service is described in this blog post. We will discuss simplified version of this analysis.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Getting the Data

First, we need get the metadata for CORD papers that we will be working with.

NOTE: We do not provide a copy of the dataset as part of this repository. You may first need to download the `metadata.csv` file from this dataset on Kaggle. Registration with Kaggle may be required. You may also download the dataset without registration from here, but it will include all full texts in addition to metadata file.

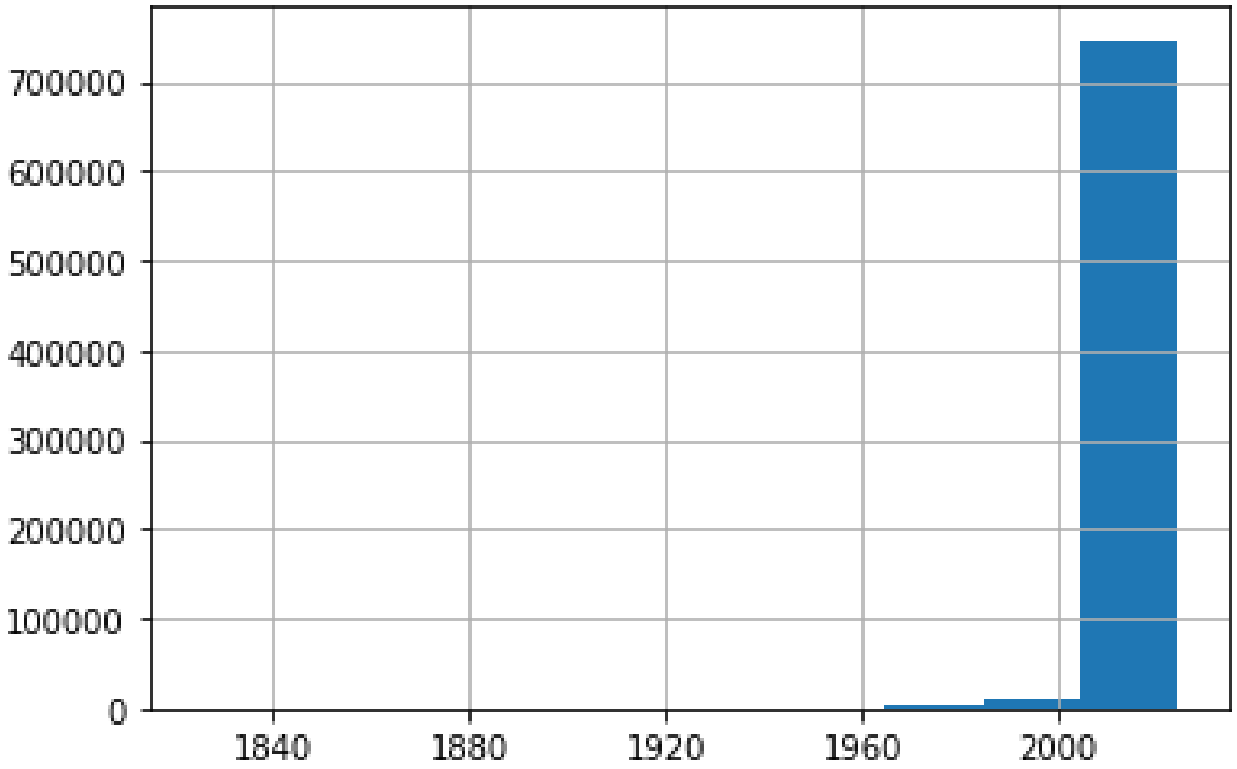
We will try to get the data directly from online source, however, if it fails, you need to download the data as described above. Also, it makes sense to download the data if you plan to experiment with it further, to save on waiting time.

> **NOTE** that dataset is quite large, around 1 Gb in size, and the following line of code can take a long time to complete! (~5 mins)

```
df = pd.read_csv("https://datascience4beginners.blob.core.windows.net/cord/metadata.csv.zip",compression='zip')
# df = pd.read_csv("metadata.csv")
df.head()
```

We will now convert publication date column to `datetime`, and plot the histogram to see the range of publication dates.

```
df['publish_time'] = pd.to_datetime(df['publish_time'])
df['publish_time'].hist()
plt.show()
```



Interestingly, there are coronavirus-related papers that date back to 1880!

Structured Data Extraction

Let's see what kind of information we can easily extract from abstracts. One thing we might be interested in is to see which treatment strategies exist, and how they evolved over time. To begin with, we can manually compile the list of possible medications used to treat COVID, and also the list of diagnoses. We then go over them and search corresponding terms in the abstracts of papers.

```
medications = [
    'hydroxychloroquine', 'chloroquine', 'tocilizumab', 'remdesivir', 'azithromycin',
    'lopinavir', 'ritonavir', 'dexamethasone', 'heparin', 'favipiravir', 'methylprednisolone']
diagnosis = [
    'covid','sars','pneumonia','infection','diabetes','coronavirus','death'
]

for m in medications:
    print(f" + Processing medication: {m}")
    df[m] = df['abstract'].apply(lambda x: str(x).lower().count(' '+m))

for m in diagnosis:
    print(f" + Processing diagnosis: {m}")
    df[m] = df['abstract'].apply(lambda x: str(x).lower().count(' '+m))
```

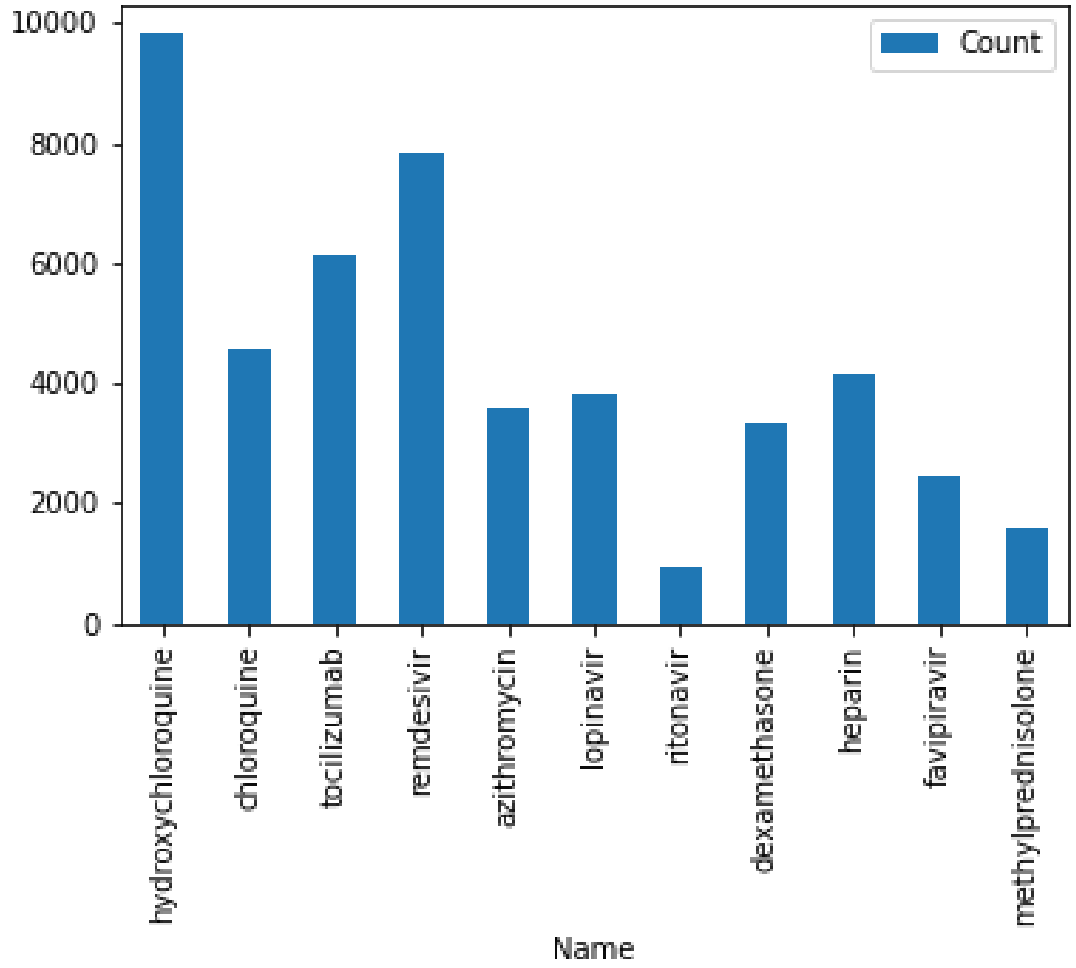
We have added a bunch of columns to our dataframe that contain number of times a given medicine/diagnosis is present in the abstract.

> **Note** that we add space to the beginning of the word when looking for a substring. If we do not do that, we might get wrong results, because *chloroquine* would also be found inside substring *hydroxychloroquine*. Also, we force conversion of abstracts column to `str` to get rid of an error - try removing `str` and see what happens.

To make working with data easier, we can extract the sub-frame with only medication counts, and compute the accumulated number of occurrences. This gives is the most popular medication:

```
dfm = df[medications]
dfm = dfm.sum().reset_index().rename(columns={ 'index' : 'Name', 0 : 'Count'})
dfm.sort_values('Count',ascending=False)
```

```
dfm.set_index('Name').plot(kind='bar')
plt.show()
```



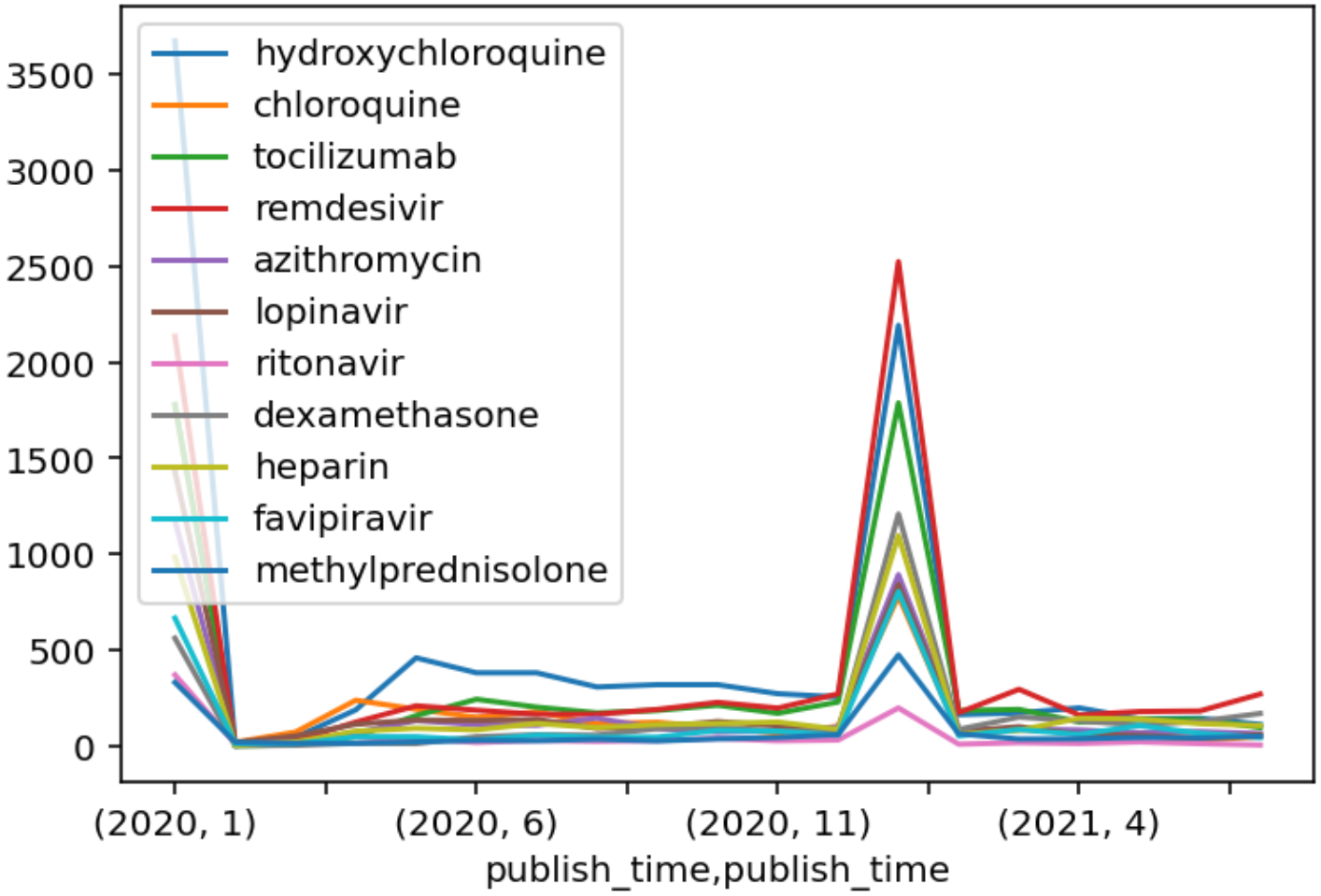
Looking for Trends in Treatment Strategy

In the example above we have `sum`ed all values, but we can also do the same on a monthly basis:

```
dfm = df[['publish_time']+medications].set_index('publish_time')
dfm = dfm[(dfm.index>="2020-01-01") & (dfm.index<="2021-07-31")]
dfmt = dfm.groupby([dfm.index.year,dfm.index.month]).sum()
dfmt
```

This gives us a good picture of treatment strategies. Let's visualize it!

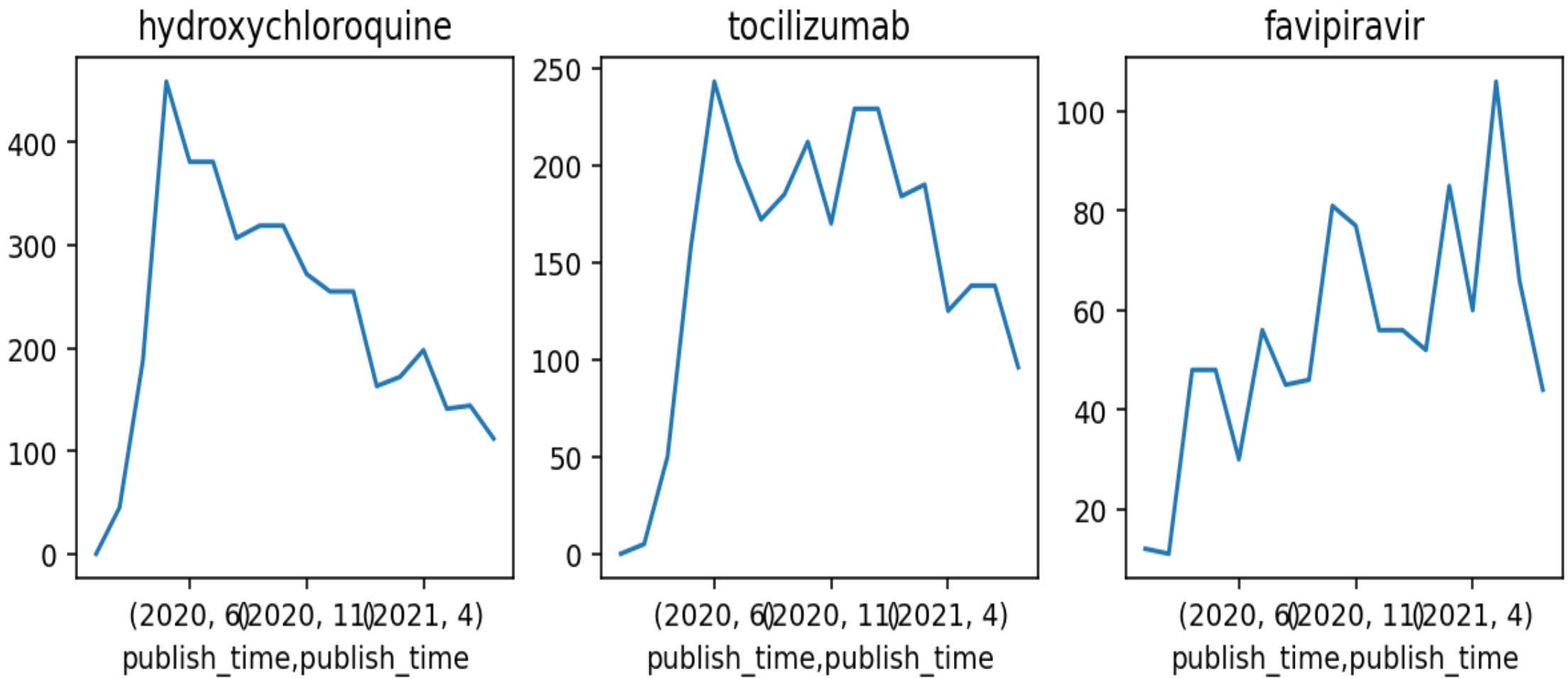
```
dfmt.plot()
plt.show()
```



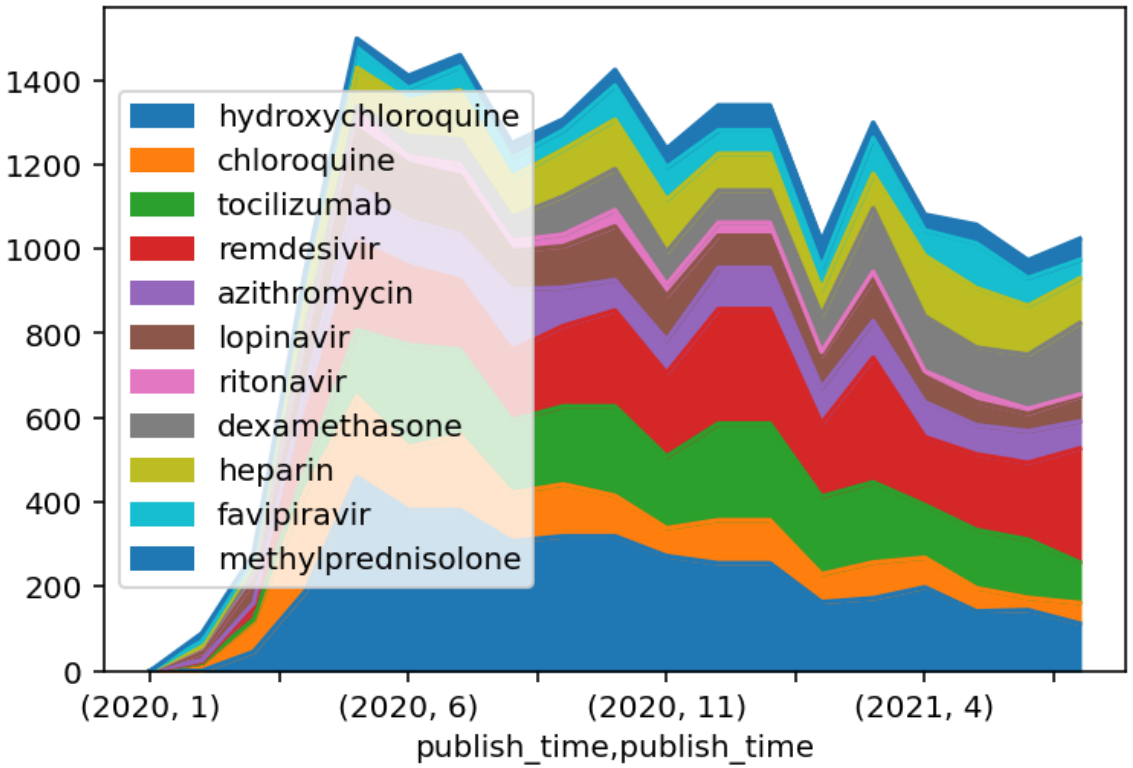
An interesting observation is that we have huge spikes at two locations: January, 2020 and January, 2021. It is caused by the fact that some papers do not have a clearly specified data of publication, and they are specified as January of the respective year.

To make more sense of the data, let's visualize just a few medicines. We will also "erase" data for January, and fill it in by some medium value, in order to make nicer plot:

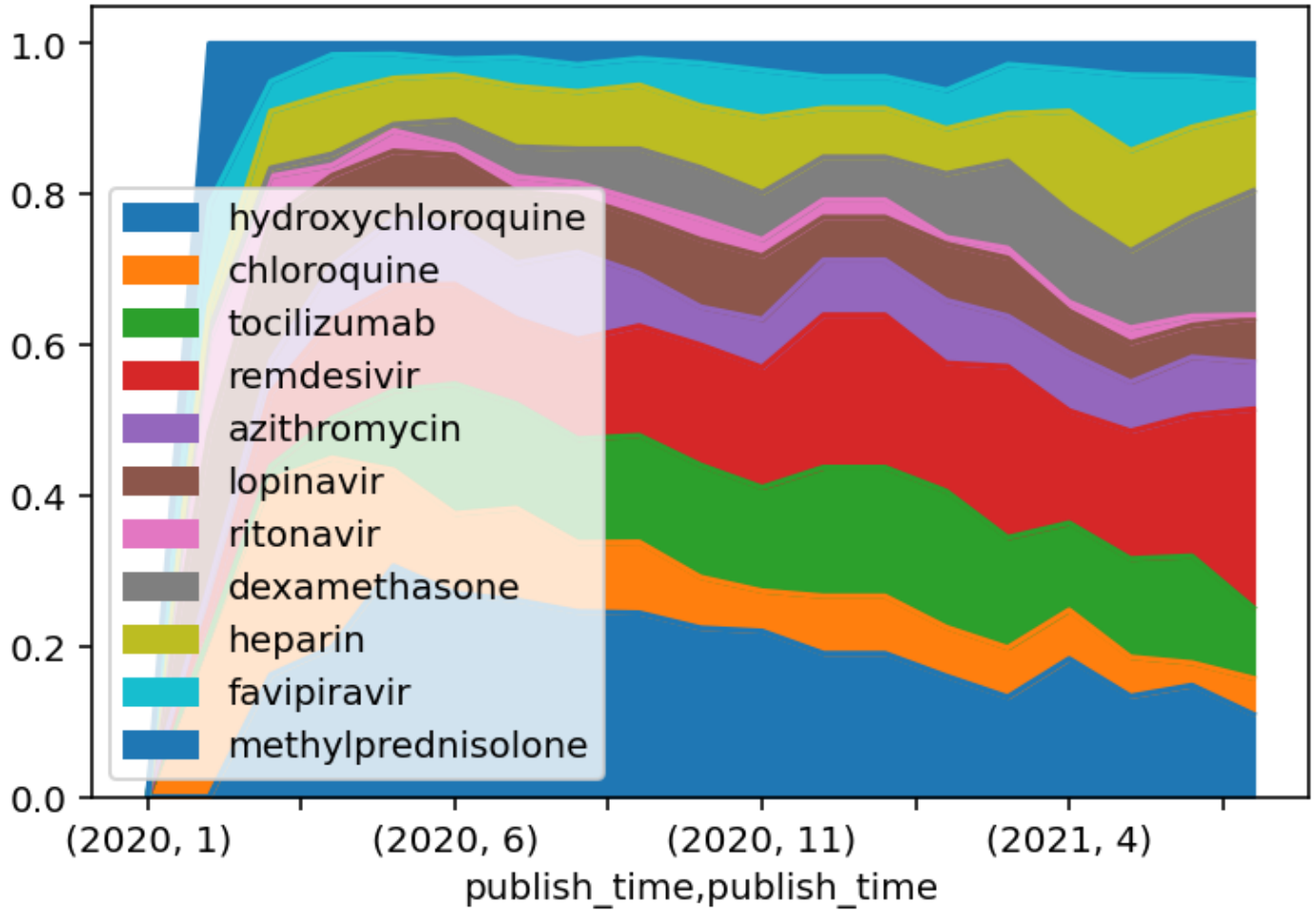
```
meds = ['hydroxychloroquine','tocilizumab','favipiravir']
dfmt.loc[(2020,1)] = np.nan
dfmt.loc[(2021,1)] = np.nan
dfmt.fillna(method='pad',inplace=True)
fig, ax = plt.subplots(1,len(meds),figsize=(10,3))
for i,m in enumerate(meds):
    dfmt[m].plot(ax=ax[i])
    ax[i].set_title(m)
plt.show()
```


hydroxychloroquine
400
300
200
100
0
(2020, 6)
(2020, 11)
(2021, 4)
publish_time,publish_time
tocilizumab
250
200
150
100
50
0
(2020, 6)
(2020, 11)
(2021, 4)
publish_time,publish_time
favipiravir
100
80
60
40
20
(2020, 6)
(2020, 11)
(2021, 4)
publish_time,publish_time

Observe how popularity of hydroxychloroquine was on the rise in the first few months, and then started to decline, while number of mentions of favipiravir shows stable rise. Another good way to visualize relative popularity is to use **stack plot** (or **area plot** in Pandas terminology):



Even further, we can compute relative popularity in percents:



Computing Medicine-Diagnosis Correspondence

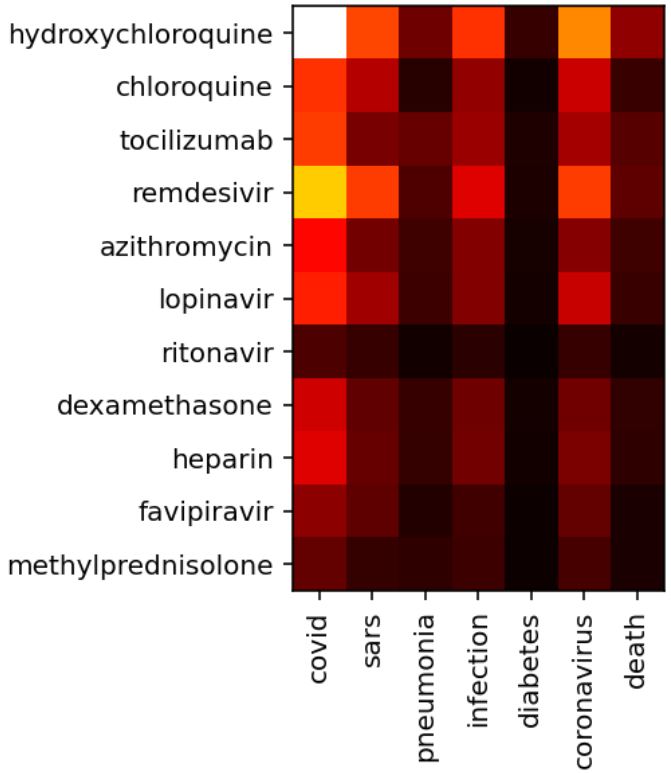
One of the most interesting relationships we can look for is how different diagnoses are treated with different medicines. In order to visualize it, we need to compute **co-occurence frequency map**, which would show how many times two terms are mentioned in the same paper.

Such a map is essentially a 2D matrix, which is best represented by **numpy array**. We will compute this map by walking through all abstracts, and marking entities that occur there:

```
m = np.zeros((len(medications),len(diagnosis)))
for a in df['abstract']:
    x = str(a).lower()
    for i,d in enumerate(diagnosis):
        if ' '+d in x:
            for j,me in enumerate(medications):
                if ' '+me in x:
                    m[j,i] += 1
```

One of the ways to visualize this matrix is to draw a **heatmap**:

```
plt.imshow(m,interpolation='nearest',cmap='hot')
ax = plt.gca()
ax.set_yticks(range(len(medications)))
ax.set_yticklabels(medications)
ax.set_xticks(range(len(diagnosis)))
ax.set_xticklabels(diagnosis,rotation=90)
plt.show()
```



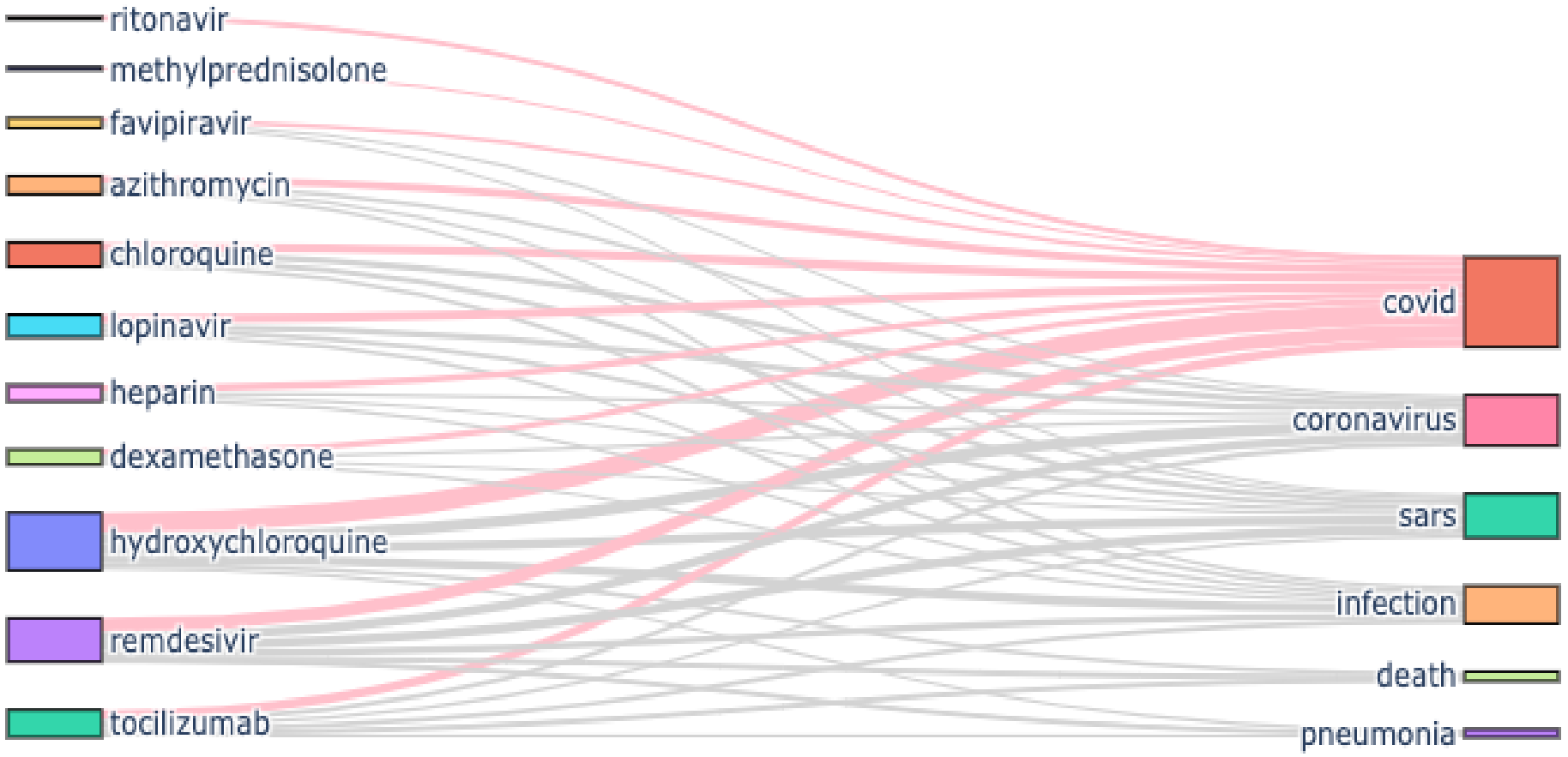
However, even better visualization can be done using so-called **Sankey** diagram! `matplotlib` does not have built-in support for this diagram type, so we would have to use Plotly as described in this tutorial.

To make plotly sankey diagram, we need to build the following lists:

- List `all_nodes` of all nodes in the graph, which will include both medications and diagnosis
- List of source and target indices - those lists would show, which nodes go to the left, and which to the right part of the diagram
- List of all links, each link consisting of:
    - Source index in the `all_nodes` array
    - Target index
    - Value indicating strength of the link. This is exactly the value from our co-occurence matrix.
    - Optionally color of the link. We will make an option to highlight some of the terms for clarity

Generic code to draw sankey diagram is structured as a separate `sankey` function, which takes two lists (source and target categories) and co-occurence matrix. It also allows us to specify the threshold, and omit all links that are weaker than that threshold - this makes the diagram a little bit less complex.

```
import plotly.graph_objects as go

def sankey(cat1, cat2, m, treshold=0, h1=[], h2=[]):
    all_nodes = cat1 + cat2
    source_indices = list(range(len(cat1)))
    target_indices = list(range(len(cat1),len(cat1)+len(cat2)))

    s, t, v, c = [], [], [], []
    for i in range(len(cat1)):
        for j in range(len(cat2)):
            if m[i,j]>treshold:
                s.append(i)
                t.append(len(cat1)+j)
                v.append(m[i,j])
                c.append('pink' if i in h1 or j in h2 else 'lightgray')

    fig = go.Figure(data=[go.Sankey(
        # Define nodes
        node = dict(
        pad = 40,
        thickness = 40,
        line = dict(color = "black", width = 1.0),
        label =  all_nodes),

        # Add links
        link = dict(
        source =  s,
        target =  t,
        value =  v,
        color = c
    ))])
    fig.show()

sankey(medications,diagnosis,m,500,h2=[0])
```


ritonavir
methylprednisolone
favipiravir
azithromycin
chloroquine
lopinavir
heparin
dexamethasone
hydroxychloroquine
remdesivir
tocilizumab
covid
coronavirus
sars
infection
death
pneumonia

Conclusion

You have seen that we can use quite simple methods to extract information from non-structured data sources, such as text. In this example, we have taken the existing list of medications, but it would be much more powerful to use natural language processing (NLP) techniques to perform entity extraction from text.