Ed Silantyev Follow
Oct 13, 2017 · 9 min read · Listen

Cryptocurrency Data Analysis Part III: Backtesting, Evaluating and Optimising a Trading Strategy

In the last tutorial we have scraped, saved and analysed bulk crypto data. Hopefully, you were able to see the beauty of combining different Python libraries to manipulate, analyse and visualise data. In this tutorial we will backtest, evaluate and attempt to optimise a trading strategy based on a simple moving average (SMA) crossover.

Loading Data

Given that you already know how to load price data for any asset on Poloniex, let's refresh how we download and store data as a pandas dataframe:

```
1  def CryptoData(symbol, frequency):
2      #Params: String symbol, int frequency = 300,900,1800,7200,14400,86400
3      #Returns: df from first available date
4      url ='https://poloniex.com/public?command=returnChartData&currencyPair='+symbol+'&end=999999999
5      df = pd.read_json(url)
6      df.set_index('date',inplace=True)
7      return df
```

CryptoData.py hosted with ❤ by GitHub view raw

Now we can go ahead and import the modules that we will use and specify that our graphs are to plotted inline:

```
import seaborn as sns
%matplotlib inline
```

Now we need to decide an asset which we will test our strategy on. Many crypto traders prefer to see their net-worth be marked to market in BTC as opposed to the greenback and hence sometimes, their primary goal is to increase amount of bitcoins they are worth. This is exactly what we will be doing; we will be testing a strategy on `BTC_LTC` crypto pair, attempting to increase our bitcoin balance. We will take 5-minute granularity data, which mean passing `300` as `frequency` argument.

```
df = CryptoData(symbol = 'BTC_LTC', frequency = 300)
```

Simple Moving Average Crossover

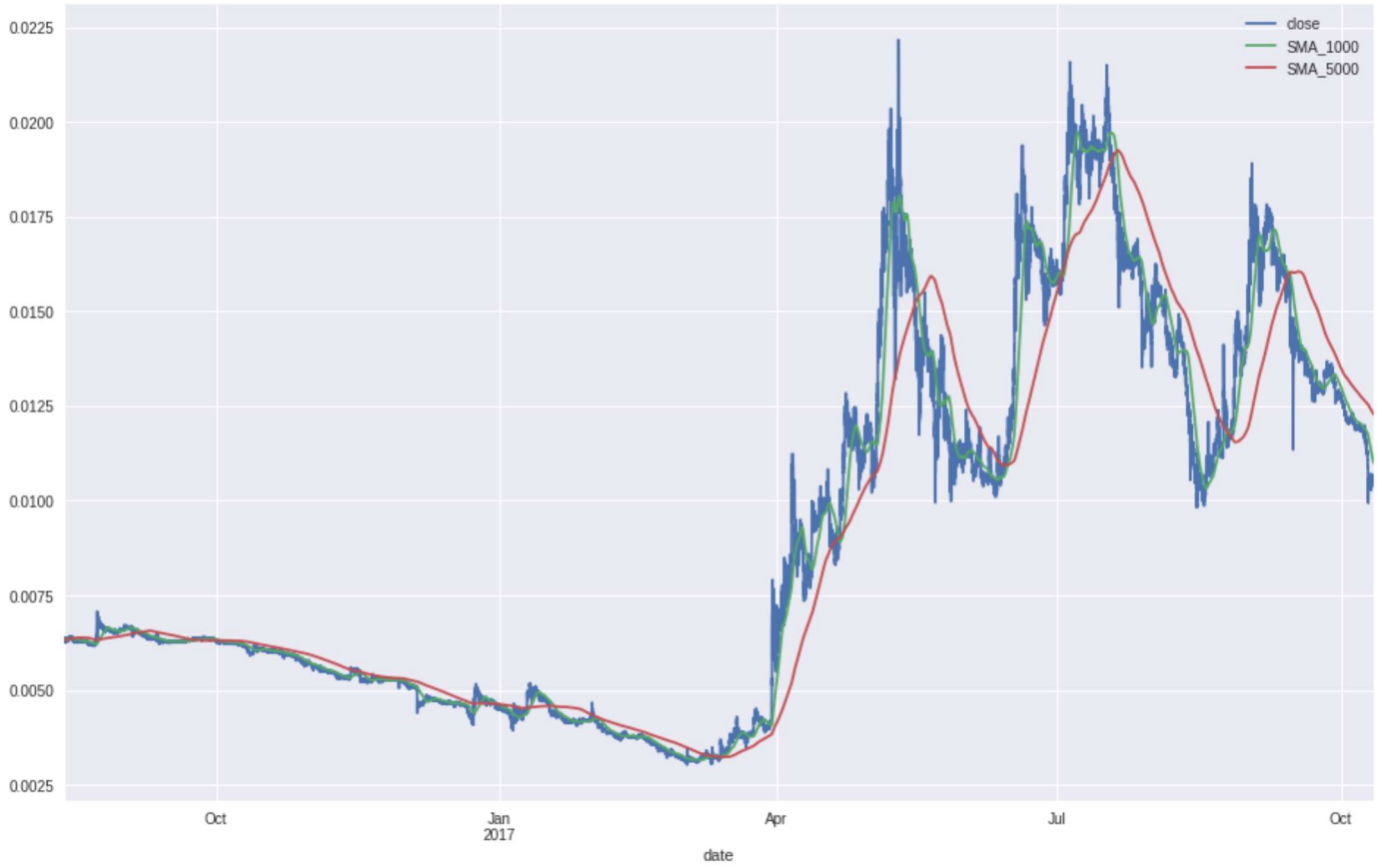
SMA is defined by one parameter — its look-back period. We calculate it as follows:

$$SMA = \frac{p_M + p_{M-1} + \cdots + p_{M-(n-1)}}{n} = \frac{1}{n}\sum_{i=0}^{n-1} p_{M-i}$$

Source: Wikipedia

Whereby we sum prices over a defined look-back period and then divide it by that look-back period. It is generally used to 'smooth' the price data, and many people argue that it gives a 'true' price because it averages upward and downward spikes in price. SMA crossover strategy consists of a leading and a lagging simple moving averages. Leading SMA has a shorter look-back period that lagging moving average. Hence, by definition, leading SMA will be more sensitive t… …e moves; lagging SMA will be slower to react. To visualise how SM… …to price, let's proceed to plot prices together with leading and lagging SMAs:

Note how easy it is to string together multiple calls to series objects to create rolling series with pandas. We arbitrarily chose 1000 and 5000 minute look-back windows. When we plotted the series, we specified to plot from 270000th observation onward and specified figure size to be 16 by 10 inches. Output:



You can now see how leading and lagging indicators react to movements in price. As mentioned, 1000-minute average reacts to recent changes quicker than the 5000-minute one. The SMA crossover strategy logic is as follows:

1. **BUY** if *Leading* SMA is **above** *Lagging* SMA by some threshold.
2. **SELL** if *Leading* SMA is **below** *Lagging* SMA by some threshold.

Threshold is applied to filter out weak signals and will be set by default as 2.5% of the

We will test our strategy by writing a function that will take in a dataframe, lead and lag look-backs and the threshold and spit back a dataframe with strategy implemented:

Now, let's take it apart. We begin by making a copy of the dataframe that we take as input. Then, we define lead and lag SMAs and we drop the rows for which SMA values are

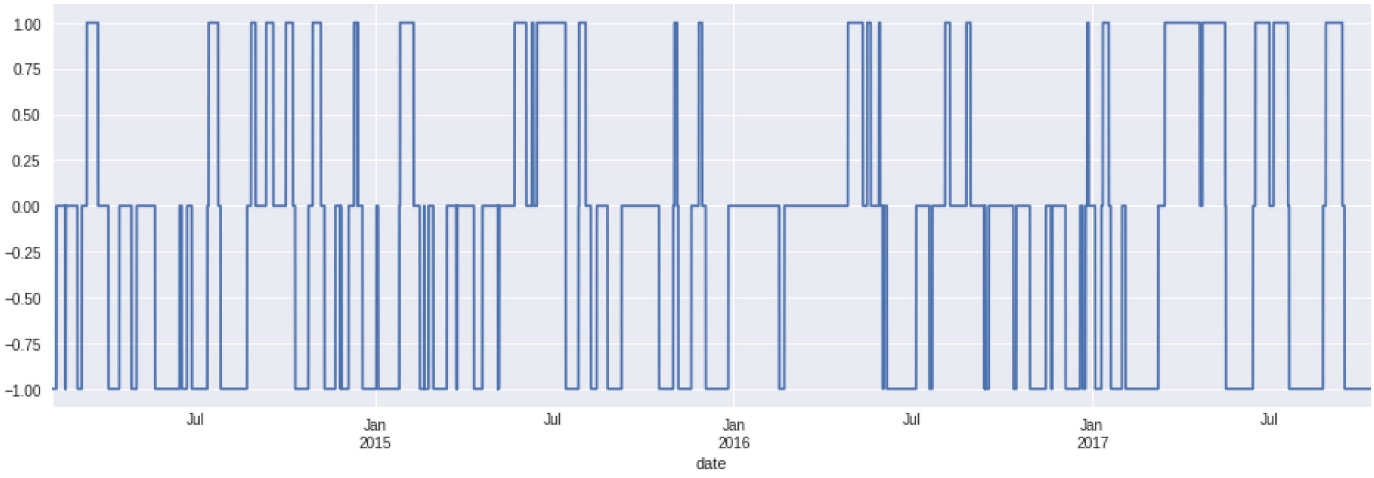

we are short and 0 means no position. We take advantage on `np.where()` function that lets us specify a predicate and conditional values, using which it will output a corresponding array. We define our `Market` column as log returns of price series. We compute our `Strategy` returns by multiplying `regime` (shifted forward to match the `Market` column). Finally, we perform a cumulative sum operation as well as apply an exponent on `Market` and `Strategy` log returns in order to recover the original normalised series. Let's test the function:

```
ma_df = test_ma(df, 1000, 5000).dropna()
```

We can plot the series of `regime` to see where we would have been long and short given our set of rules:

```
ma_df['regime']plot(figsize=(16,5))
```

Will output the following:



We make plenty of long and short trades throughout the period. To see whether we

```
ma_df[['Market','Strategy']].iloc[-1]
```

And we can see that not only have we out-performed the market, but we have done so by a factor of at least 20 (normalised series start from 1):

```
Market       0.422360
Strategy    10.384434
Name: 2017-10-11 13:10:00, dtype: float64
```

To visualise how we fared against the market, let's plot the `Market` and `Strategy` series since January 2016:

```
ma_df[['Market','Strategy']][200000:].plot(figsize = (16,10))
```

Output:

We can see that while the market went sideways for a considerable time, we managed to take advantage of these fluctuations by buying and selling according to SMA crossover rules. It is important to note that most profit was made during the so-called “alt-coin boom” during which many altcoins took off in value.

Optimising and Visualising Strategy Parameters

We are at a good starting point — we have achieved x10 backtest returns by using arbitrary lead and lag look-backs. Can we do better? Of course we can! We will now brute-force through different combinations of lead and lag look-back periods, saving final profit and loss (PnL) for each such combination. At the end, we will plot the heatmap of these PnLs as function of period combinations.

We start by defining arrays containing integers corresponding to leading and lagging look-back windows respectively. `np.arange()` is a perfect candidate for the task as it will take the sequence bounds and interval to produce ordered NumPy arrays. We will then use a list comprehension to create an array of `[lead,lag]` pairs. Finally we will define a dataframe where we will store our final PnLs:

```
leads = np.arange(100, 4100, 100)
lags = np.arange(4100, 8100, 100)
lead_lags = [[lead,lag] for lead in leads for lag in lags]
pnls = pd.DataFrame(index=lags,columns = leads)
```

If we run `len(lead_lags)` we will see that we have to make `1600` backtests to fill up the `pnls` matrix. This will take some time. In order to fill up this matrix we need to:

1. Loop over the `lead_lags` array.
2. Take the final PnL and save it in `pnls` matrix.
3. Repeat until there are no more candidates left in `lead_lags` .

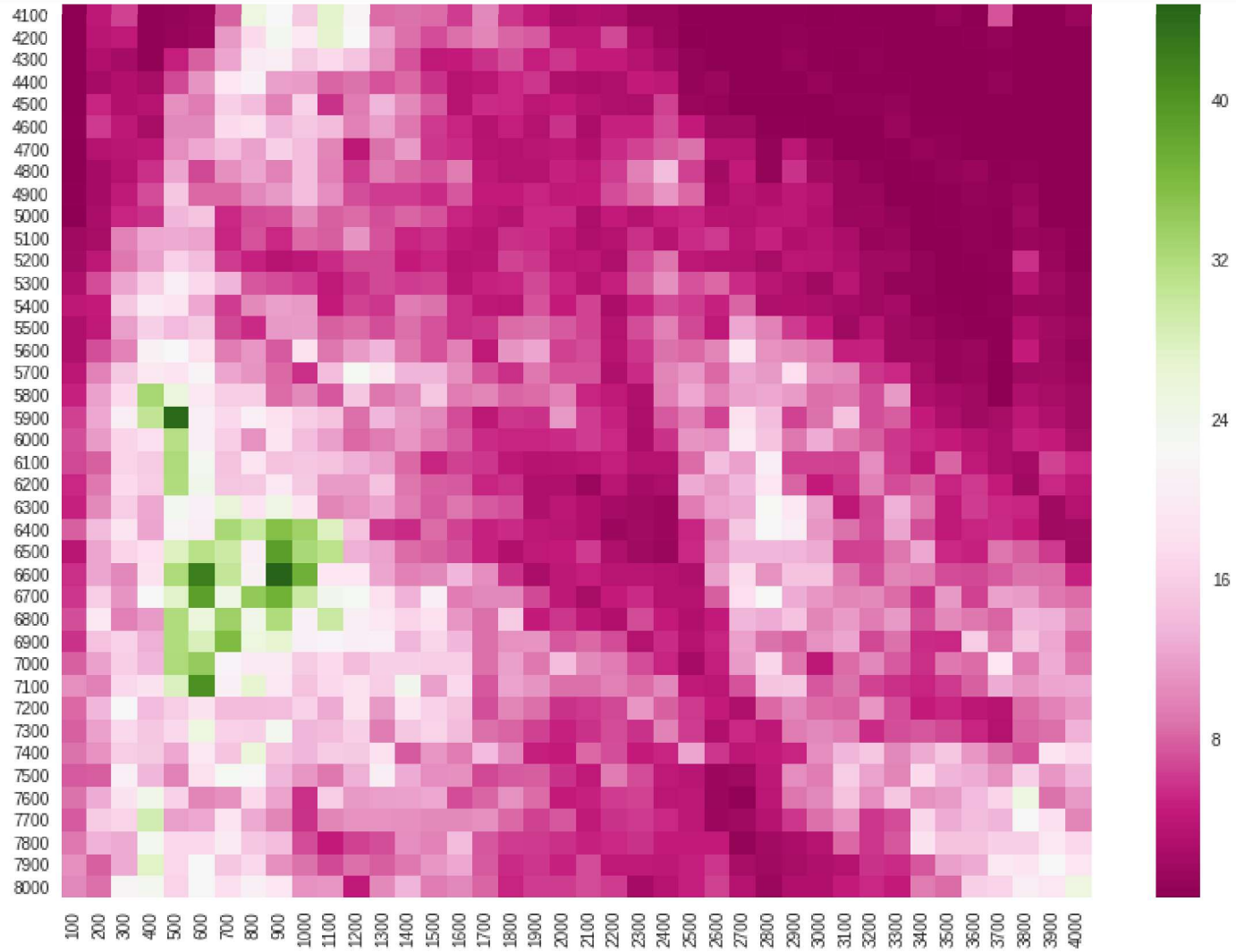

You should see the stage of the progress at each iteration:

```
100 4100 0.109696350799
100 4200 0.168002440847
100 4300 0.129377447402
100 4400 0.10318276452
100 4500 0.165473622191
100 4600 0.250176153044
100 4700 0.17482270303
100 4800 0.192016648903
100 4900 0.261229591629
100 5000 0.182849263658
100 5100 1.4943823881
100 5200 1.94480944908
100 5300 2.71494860071
100 5400 3.88657722516
100 5500 2.70262997391
100 5600 2.75105468698
...
```

When the loop finishes, we are ready to visualise `pnls` matrix. We will use Seaborn's `heatmap()` to do so. Before that, we need to turn the members of our matrix into `float` numbers so that Seaborn can plot them. We will also make our plot 14 by 10 inches:

```
PNLs = pnls[pnls.columns].astype(float)
plt.subplots(figsize = (14,10))
sns.heatmap(PNLs,cmap='PiYG')
```

We get a nice looking heatmap that can help explain which combinations of SMA lookbacks work best:

Looking at the visualisation, we can see a green patch in the lower left corner, perhaps signifying that the ratio of lead to lag lookback of 1/7 to 1/8 is most optimal. We can now proceed to find the lookback periods that give the best PnL analytically:

```
PNLs.max()
```

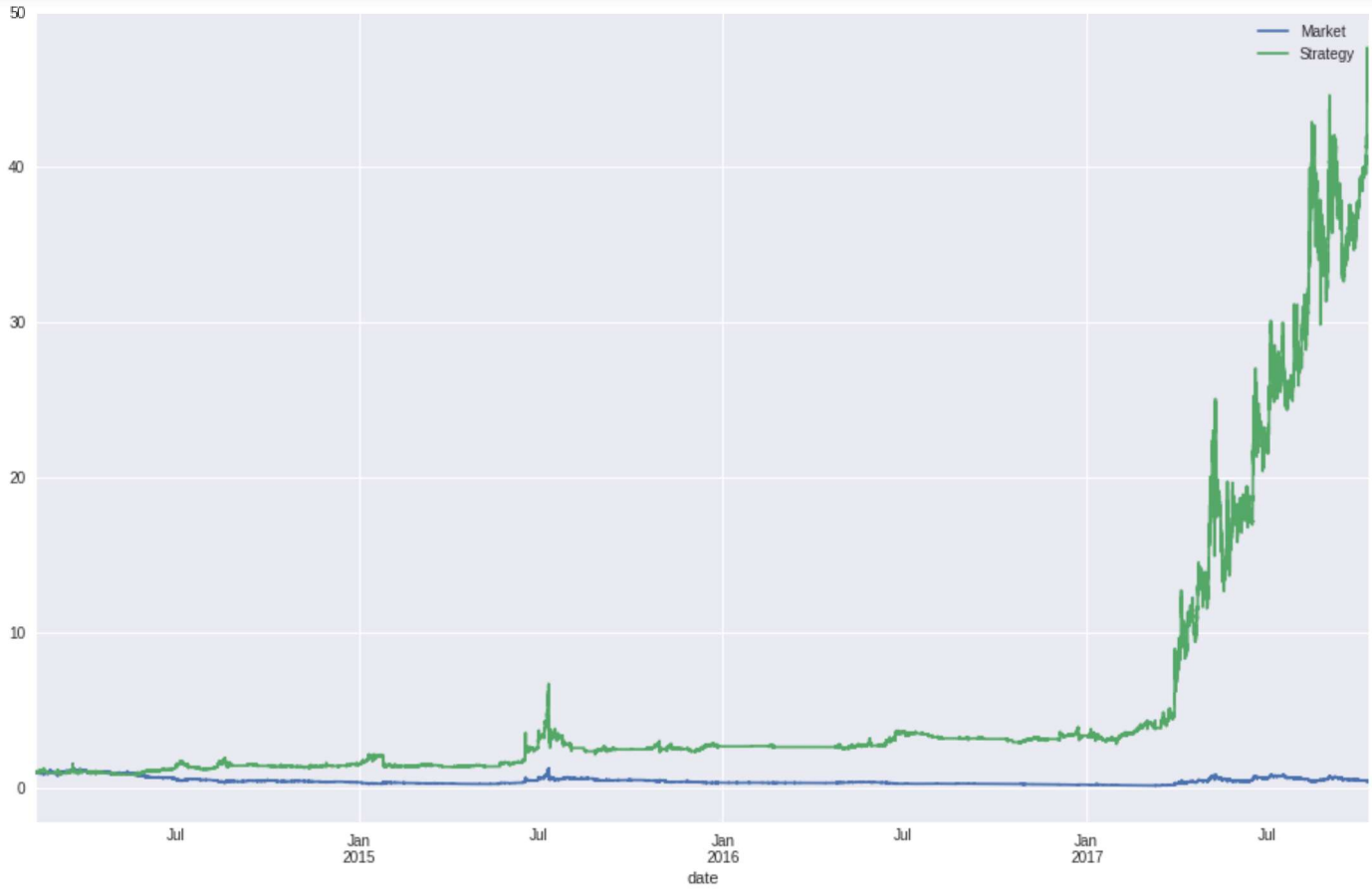
We get series of `lead` lookbacks and corresponding maximum PnL values:

```
100      10.864477
```

```
700     35.842106
800     34.182762
900     44.826745
1000    37.054376
1100    30.945689
1200    22.989689
1300    20.970539
1400    23.822840
1500    20.404080
1600    17.185955
1700    13.923057
1800    15.732017
1900    12.414719
2000    14.433515
2100    10.450186
2200     9.771460
2300    11.302012
2400    14.350452
2500    12.639792
2600    14.260199
2700    19.634837
2800    22.676327
2900    20.287311
3000    14.714964
3100    15.916137
3200    15.177753
3300    15.109187
3400    18.476793
3500    15.270295
3600    16.246062
3700    19.790292
3800    24.930189
3900    22.171754
4000    24.840235
dtype: float64
```

Lead of 900 periods produces maximum PnL. By referencing back the the matrix heatmap, we can see that corresponding lag period is `6600`. Let's confirm it by running `PNLs[900][6600]`; and indeed we get `44.826745`.

Now that we have found optimal parameters for the strategy let's visualise it:

4400% is a hefty return. Remember that `BTC` is base currency in our case, so the dollar gain will be even greater in percentage terms, given recent price appreciation. Next up, we will talk about the assumptions that we have made in our backtests. These assumptions lead to subtle biases that will affect live trading performance.

Subtleties

We have all heard the saying:

> *"Assumption is the mother of all f***-ups"*

And it could not have been more true in the world of algorithmic trading.

Transaction Costs

Commissions. We have assumed no transaction costs, even though typical exchanges

Shorting. We assume that we can openly short a cryptocurrency pair and that we pay no fees for holding short positions. In reality, some exchanges do not support shorting and if they do, other fees are associated with such transactions.

Slippage. Another assumption is that we can always get filled on the close price. Given how 'thin' some crypto pairs books are, other things being equal, we will get filled at progressively worse prices as our positions grow in size. In addition, as other traders may use similar signals, it will only increase the chances that the price may "run away" from us as we try and get a fill.

Market Impact. In our backtest, we assume that our trades have no impact on subsequent market dynamics. In reality, market can react positively or negatively to a trade. Backtesting market impact creates a never ending spiral of complexity, as it depends, upon other things on liquidity, number of market participants and different states of the market.

Biases

Overfitting. When we optimised for the best possible combination of leading and lagging look-back periods, we have taken the available historical data and threw a bunch of numbers at it to see what sticks. Whilst we did find a pattern that suggested that best PnLs are the ones whose lead / lag ratio is around 1/8, we ultimately did that on historical data and there is no guarantee that the same results would hold for live performance. In order to overcome this phenomenon, we could split our data into two sets — the one we find the best parameters on and the one we test these parameters on. If the test PnL holds up, it is safe to assume that the parameters are significant. There is a whole study in Statistics dedicated primarily to mitigation of overfitting.

Exchange Risk

Last but definitely not least, it is almost impossible to model exchange risk. Historically, a large portion of exchanges get hacked or otherwise compromised. Finding trustworthy exchanges requires further research.


libraries and different hacks, such as list comprehensions.

I hope you continue to find these tutorials useful. Give this article some CLAPS, so that other people can come across and find it useful too!