BacktraderCon 2017

When: Saturday 3rd and Sunday 4th of June

Where: Munich, Germany

Who: Mr. backtrader himself and two other fellow backtraders

(the 1st user eve also shared some minutes with the team)

BacktraderCon 2017 – The Summary

- Being this such a great event it was decided it should take place at the same time other great events happening, namely:
  - The Champions League Final won by Real Madrid on Saturday. The 1st time a team wins it 2 years in a row (since it is called Champions League)
  - An Italian rider (Dovizioso) winning the MotoGP race in Italy (Mugello) with an Italian Bike Ducati (last time was over 40 years ago) on Sunday

- The Con produced its own set of great events:
  - Completing the 1st data feed/broker implementation to trade CryptoCurrencies with Kraken
  - Eating real not-for-tourists Bavarian specialties

- A GitHub repo hosts the small snippets that were done during the event
  - https://github.com/mementum/backtrader.con

__init__ vs next

- During `__init__` you create **lazy** objects, which you can later access during `next`

```
class MyStrategy(bt.Strategy):
    def __init__(self):
        self.hilodiff = self.data.high - self.data.low
        self.mysma = bt.ind.SMA()

    def next(self):
        if self.hilodiff[0] > 20.0:
            print('greater than 20')
        if self.mysma[0] – self.data.close[0] > 5:
            print('sma is 5 units away from close')
```

- `hilodiff` and `mysma` were created in `__init__`, the first through a deceptively simple arithmetic operation and the second by creating an indicator
- And the current moment `[0]` of both is accessed during `next`. Because this would break:

```
class MyStrategy(bt.Strategy):
    def __init__(self):
        self.hilodiff = self.data.high - self.data.low
        if self.hilodiff[0] > 20.0:  # WARNING … DON'T DO THAT DURING INIT
            print('greater than 20')
```

- During `__init__` data feeds are empty … there is no `[0]` … (if your code isn't breaking … you are preloading data and you are being deceived by Python array arithmetic …)

Using [0] (or [x]) or NOT

- `[x]` gives you the possibility to determine which moment of the stream you access

```
def next(self):
    if self.hilodiff[0] > 20.0:  # ACCESSING CURRENT MOMENT … index 0
        print('greater than 20')
```

- If `[x]` is not used in the '>' comparison … the CURRENT MOMENT `[0]` is implied

```
def next(self):
    if self.hilodiff > 20.0:  # ACCESSING CURRENT MOMENT … by entirely omitting the operator []
   print('greater than 20')
```

- Remember … in backtrader the previous values are accessed with negative indices (the **last** value produced before the current moment is at index `-1` and this is Pythonic … `-1` always means the **last** in Python)
- But ATTENTION … this may break … if it doesn't you are preloading data and being deceived by Python and array arithmetic

```
def next(self):
    if self.hilodiff[-1] > 20.0:  # THIS COULD BREAK … we haven't told the strategy we need to look backwards
        print('greater than 20')
```

- Before carrying on … consider that a live feed (or dynamic if pulling from a database) may not be preloaded and there is only 1 point (*backtrader* also has a feature to save memory reducing buffers … and unless something is done … buffers will be reduced to size `1`, the minimum)

__init__ indexing () vs next indexing []

- During `__init__` you can ask for previous values … indexing with `(x)`

```
def __init__(self):
    self.hi_vs_old_hi = self.data.high – self.data.high(-5)  # current minus the value 5 periods ago
```

- From the previous slides: THIS CREATES AN OBJECT … is empty, so this will likely break

```
def __init__(self):
    self.hi_vs_old_hi = self.data.high – self.data.high(-5)  # current minus the value 5 periods ago
    if self.hi_vs_old_hi[0] > 20.0:  # REMEMBER … if it doesn't break … PRELOADING IS DECEIVING YOU
        print('something')
```

- When working in next … it will work

```
def next(self):
    if self.hi_vs_old_hi[0] > 10:
        print('the current high is really higher')
```

- And even looking backwards

```
if self.hi_vs_old_hi[-5] > 10:
        print('the current high is really higher')
```

- BUT TOO FAR AWAY … it may break again

```
if self.hi_vs_old_hi[-6] > 10:  # WE ONLY ASKED FOR self.data.high(-5), -6 MAY NOT BE THERE
        print('the current high is really higher')
```

- Because it will either break with this error or produce false results

```
NOT PRELOADING DATA: IndexError: array index out of range
PRELOADING DATA: -5 will end up looking at the latest values of the preloaded data and deducted it from the 1st.
```

__init__ indexing () vs next indexing [] … more

- Why not simply doing this directly during `next` and forget the `(-5)` in __init__

```
def next(self):
    if self.hi[0] – self.data.high[-5] > 10:  # BECAUSE THIS MAY BREAK
        print('the current high is really higher')
```

- Because it will either break with this error or produce false results

```
NOT PRELOADING DATA: IndexError: array index out of range
PRELOADING DATA: -5 will end up looking at the latest values of the preloaded data and substract it from the 1st.
```

- Live data feeds may for example save memory and not buffer the last 5 periods … unless the `(-5)` during `__init__` has explicitly requested that

__init__ indexing () vs next indexing [] … wrap

- Why it breaks when the data is not preloaded?

  Because the strategy goes immediately into next and there is no data for 5 bars ago

- Why does it produce false results when the data is preloaded?

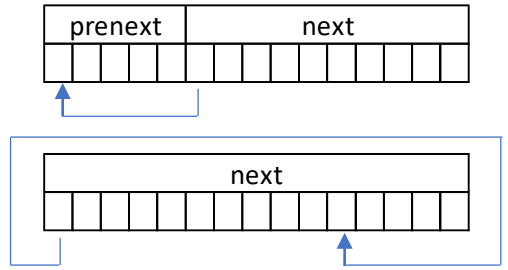
  Because the strategy goes immediately into next and the -5 wraps over to the end of the array … because there is still nothing 5 bars ago … but Python array indexing wraps over to the end of the array

- But why does it work with (-5)?

```
def __init__(self):
    self.hi_vs_old_hi = self.data.high – self.data.high(-5)
```

  During `__init__` this (-5) creates an object which delivers the values from 5 bars ago … and this object tells the strategy … PLEASE WAIT 5 bars before you go into `next`

- See the difference when going first through `prenext` or straight to `next`
  - 5 periods have been consumed in `prenext` thanks to the `(-5)` indication
  - There is data buffered to later get to it with `[-5]`
  - But with no indication … the arithmetic wraps over and gives us something else
  - Or it may break if not data has been preloaed



- Adding other “buffer” (or period) demanding things, will give us more buffer

```
def __init__(self):
    self.hi_vs_old_hi = self.data.high – self.data.high(-5)
    self.mysma = bt.ind.SMA(self.data, period=30)  # will automatically ask for at least 30 periods of buffering
```

Mixing TimeFrames: feeds and signals

- Yes … it's easy with *backtrader* … but the design principles have to be not only understood … also ACCEPTED
- Things are *lines* objects in *backtrader* and lines have a property: **The minimum period before delivering data**
  - This property is the one controlling when one is in `prenext` and when one can move to `next`
  - With that in mind: data feeds have an obvious minimum period: `1`
  - But `1` DIFFERENT if the timeframes are different
- What are we doing here:
  - 5-minutes signal (Crossover of Close/SMA)
  - 1-day signal (Crossover of Close/SMA)
  - Which effectively means that the daily signal controls the if the intraday signal can operate
- Now the ACCEPTANCE
  - There will be PLENTY of 5-minutes bar delivered BEFORE the strategy's `next` is CALLED a single time. Why?
  - Because many `5-minutes` moments are delivered before the `1-day` data feed can deliver for the 1st time
  - And the strategy will NOT move into `next` before all *lines* are delivering, to make sure that at least `[0]` has a value
- Of course
  - If we add an indicator, SMA with period 5 for example, … it will take LONGER to get to next, because the indicator needs 5 additional days to start delivering

Mixing TimeFrames: coding it …

- The basics … the signals for each timeframe

```
class St(bt.Strategy):
    def __init__(self):
         dsma = bt.ind.SMA(self.dnames.days, period=5)
        self.dsignal = dsignal = bt.ind.CrossOver(self.dnames.days, dsma)
        dsignal.plotinfo.plotname = 'MSignal'

        msma = bt.ind.SMA(self.dnames.minutes, period=10)
        self.msignal = msignal = bt.ind.CrossOver(msma, self.dnames.minutes)
        msignal.plotinfo.plotname = 'DSignal'
        self.osignal = bt.And(msignal, dsignal.lines[0]()) # mixing signals from 2 timeframes
```

- And now the magic to make the signals from different timeframes play together: `()` (with no arguments)
  - This is called COUPLING …
  - `(-x)` is to ask for the value from x periods ago
  - `()` is requesting the framework to make a well behaved couple out of 2 different timeframes (hence COUPLING)

```
self.osignal = bt.And(msignal, dsignal())
```

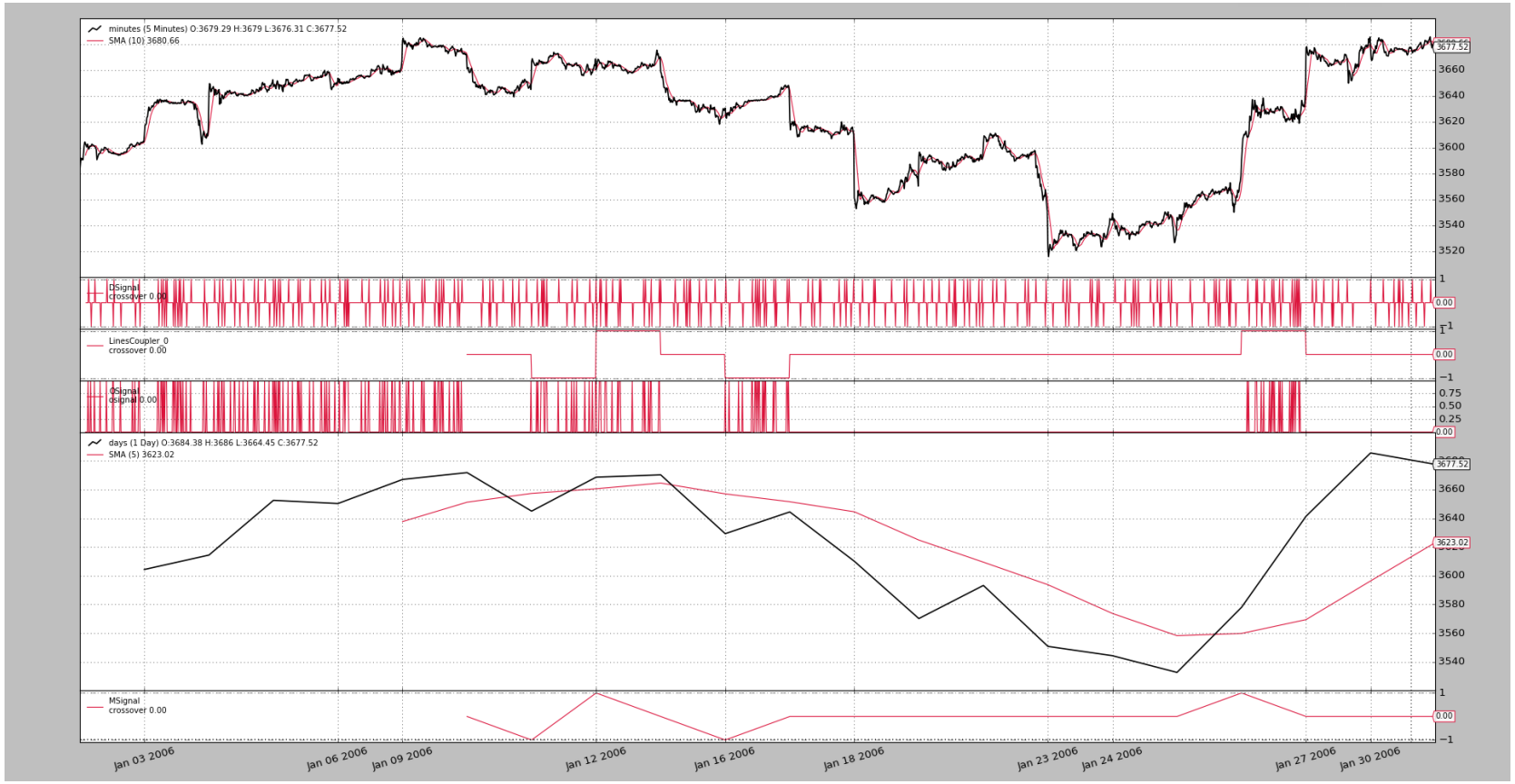
  - `osignal` will have non-zero value when both `msignal` (minutes) and `dsignal` (days) have a non-zero value.
- And it works (see the output)

Mixing TimeFrames: the output

```
Len Datetime MinLen MinTime MinClose DaysLen DaysTime DaysClose MinSignal DaysSignal OSignal
00613,2006-01-10 09:05:00,00613,2006-01-10 09:05:00,3660.84,00006,2006-01-09 23:59:59.999989,3671.78,0.00,0.00,0.00
00614,2006-01-10 09:10:00,00614,2006-01-10 09:10:00,3661.05,00006,2006-01-09 23:59:59.999989,3671.78,0.00,0.00,0.00
…
00635,2006-01-10 10:55:00,00635,2006-01-10 10:55:00,3647.29,00006,2006-01-09 23:59:59.999989,3671.78,0.00,0.00,0.00
00636,2006-01-10 11:00:00,00636,2006-01-10 11:00:00,3648.10,00006,2006-01-09 23:59:59.999989,3671.78,-1.00,0.00,0.00
00637,2006-01-10 11:05:00,00637,2006-01-10 11:05:00,3648.73,00006,2006-01-09 23:59:59.999989,3671.78,0.00,0.00,0.00
…
00714,2006-01-10 17:30:00,00714,2006-01-10 17:30:00,3644.94,00006,2006-01-09 23:59:59.999989,3671.78,0.00,0.00,0.00
00715,2006-01-11 09:05:00,00715,2006-01-11 09:05:00,3666.44,00007,2006-01-10 23:59:59.999989,3644.94,-1.00,-1.00,1.00
00716,2006-01-11 09:10:00,00716,2006-01-11 09:10:00,3668.77,00007,2006-01-10 23:59:59.999989,3644.94,0.00,-1.00,0.00
…
```

- The combination of minimum periods keeps the strategy in `prenext` until 613 5-minutes bars have been seen and also 6 of the 1-day.
- At length 636 the daily signal kicks in, but there is no minute signal and `osignal` remains with a value of 0.
- At length 715 both signals kick in (-1 and -1) and that changes `osignal` to `1`
- At 716, the daily signal goes back to 0 and even if the 5-minutes signal is -1, `osignal` goes back to 0
- `There is even a plot …`

Mixing TimeFrames: the output … plotted



- Notice 2 things
  - The `LinesCoupler` plot. This is a side effect of adapting the daily timeframe to the intraday one. The step-effect shows how the daily signal is stretched over the equivalent intraday period
  - `osignal` starts early … and that's because in Python … "1 and NaN" is True. But remember that `next` is not called for the 1st time until much later, making that spurious effect not relevant.

Mixing TimeFrames: LinesCoupler plot control

- Adapt only the needed line from crossover … as in `` `self.osignal = bt.And(msignal, dsignal.lines[0]())` ``
- Or couple separately and control the object

```
coupled = dsignal()
coupled.plotinfo.plot = False
self.osignal = bt.And(msignal, coupled)
```

Mixed timeframes with Interactive Brokers

- The obvious question is if the same strategy can be applied to real data feeds and real brokers … and “yes”
- One needs only to use the `Store` of the given provider
  - The strategy doesn’t know where the data feeds come from
  - The strategy doesn’t know which broker is executing the orders
  - The strategy has the same logic regardless of data and broker providers
- This all what’s needed to reapply the previous set of signals to Interactive Brokers

```
ibstore = bt.stores.IBStore()
cerebro.broker = ibstore.getbroker()  # Broker

data0 = ibstore.getdata(dataname='EUR.USD-CASH-IDEALPRO')
cerebro.resampledata(data0, timeframe=bt.TimeFrame.Minutes, compression=1, name='minutes')

data1 = ibstore.getdata(dataname='EUR.USD-CASH-IDEALPRO', name='days')
cerebro.resampledata(data1, timeframe=bt.TimeFrame.Minutes, compression=5, name='days’)
```

- The “Con” took place during a weekend and that’s why the timeframes were changed:
  - The IB demo was changed and it offers a restricted (and fake) amount of data
  - Working with 1-minute and 5-minutes was better
  - The names `minutes` and `days` are kept for the data feeds to keep the logic in the strategy 100% unchanged
  - The output is separated in two lines (1 per data feed) to make easier to read it
- And the output …

Mixed timeframes with Interactive Brokers …

```
$ ./ibstore-conn.py
Server Version: 76
TWS Time at connection:20170603 12:52:29 CET
Datetime, Open, High, Low, Close, Volume
***** STORE NOTIF: <error id=-1, errorCode=2104, errorMsg=Market data farm connection is OK:ibdemo>
***** STORE NOTIF: <error id=-1, errorCode=2106, errorMsg=HMDS data farm connection is OK:demohmds>
***** DATA NOTIF: DELAYED
Data0, 0026, 2017-06-03T00:25:00.000000, 1.0954, 1.0956, 1.0954, 1.0956, -1.0, 0.0
Data1, 0006, 2017-06-03T00:25:00.000000, 1.0952, 1.0956, 1.0951, 1.0956, -5.0
Data0, 0027, 2017-06-03T00:26:00.000000, 1.0956, 1.0958, 1.0955, 1.0957, -1.0, 0.0…
…
Data0, 0413, 2017-06-03T06:52:00.000000, 1.1032, 1.1032, 1.1031, 1.1032, -1.0, 0.0
Data1, 0083, 2017-06-03T06:50:00.000000, 1.1023, 1.1032, 1.1023, 1.1031, -5.0
***** DATA NOTIF: LIVE
COULD DO SOMETHING
Data0, 0414, 2017-06-03T06:53:00.000000, 1.11855, 1.11898, 1.11855, 1.11895, 0.0, 0.0
Data1, 0083, 2017-06-03T06:50:00.000000, 1.1023, 1.1032, 1.1023, 1.1031, -5.0
Data0, 0415, 2017-06-03T06:54:00.000000, 1.11896, 1.11905, 1.11893, 1.11899, 0.0, 0.0
…
```

- As before, the longer timeframe needs a period of 6 before kickstarting
- In this case, there are a lot less 1-minute bars in 5-minutes as 5-minutes in 1-day (the previous timeframes) and that’s why the 1minute Data0 is already delivering at 26
- But it works as expected … with no changes to the strategy

Replaying instead of Resampling

- Just for the sake of it … replaying capabilities to the Interactive Brokers sample were added

```
ibstore = bt.stores.IBStore(**eval('dict(' + args.store + ')'))
cerebro.broker = ibstore.getbroker()  # Broker

data0 = ibstore.getdata(dataname=args.data0)
cerebro.resampledata(data0, timeframe=bt.TimeFrame.Minutes, compression=1, name='minutes')

data1 = ibstore.getdata(dataname=args.data0)
if not args.replay:
    cerebro.resampledata(data1, timeframe=bt.TimeFrame.Minutes, compression=5, name='days')
else:
    cerebro.replaydata(data1, timeframe=bt.TimeFrame.Minutes, compression=5, name='days’)
```

- A argument `--replaydata` and the choice between resampling and replaying
- For the sake of it and making it more flexible, the store can also take arguments and the data name can also be specified
- Notice that in both samples `ibstore.getdata` is called twice. The reason being, that the resampling/replaying is done with two different timeframes, and independent data feeds will do backfilling with different timeframe requests

Yahoo Finance still lives in Backtrader

- On May 17th 2017 Yahoo discontinued (given all signs) the historical data download service (let's call it v1 API)
- With some reports in the community, some extra information from here and there, the `v7` API was found and could be implemented (v7 is part of the URL)
- The new format does some things which the previous didn't:
  - Uses a cookie from the finance service … which forces two http requests instead of one
  - Delivers the data in the right order: oldest to newest
  - Gives the prices already adjusted … and a key to recalculate the original prices
  - Sometimes introduces `nul` in the prices
- After a couple of iterations … the data feed is working again, since May 24th 2017
- No need to change any code
- Of course Yahoo may discontinue/change the current service … beware … consider using something else (Quandl has the free WIKI curated data)

Mixing Data Feeds: Yahoo and IB

- The next logical way of thinking is whether one can mix data feeds from different providers … the answer is yes …
- As before: the strategy doesn't know who's the provider or the broker.
- The code is changed to use Interactive Brokers and a Yahoo downloaded feed

```
ibstore = bt.stores.IBStore(**eval('dict(' + args.store + ')'))
cerebro.broker = ibstore.getbroker()  # Broker

data0 = ibstore.getdata(dataname='TWTR',
                        timeframe=bt.TimeFrame.Minutes, compression=5)
cerebro.resampledata(data0, timeframe=bt.TimeFrame.Minutes, compression=5, name='minutes')

data1 = bt.feeds.YahooFinanceCSVData(dataname='twtr-eomay.csv', timeframe=bt.TimeFrame.Days,)
cerebro.resampledata(data1, timeframe=bt.TimeFrame.Days, name='days')
```

- Et voilá … the same strategy works with the data, but let's add a couple of notes to the above
  - During the weekend the IB demo delivers daily data for the weekend and even for bank holidays
  - That's why TWTR daily data was downloaded from Yahoo and edited to fake the missing days

An undocumented jewel

- It may have been noticed in the samples that the data feeds in the strategy were being referenced as
  - self.dnames.minutes and self.dnames.days
- `minutes` and `days` are the names assigned in for example
  - cerebro.resampledata(data0, timeframes=bt.TimeFrame.Minutes, compression=5, name='minutes')
- Where is this in the documentation? It isn't
- The functionality originated after some discussions in the Community, specifically here
  - https://community.backtrader.com/topic/152/multi-asset-ranking-and-rebalancing/22
- The functionality got finally into the next stable release and … it never made it to the documentation
- But it is now in it: https://www.backtrader.com/docu/strategy.html

Moving into uncharted territory

Cryptocurrencies

Kraken – CryptoCurrencies - Trading

- The participants came by with an incredible amount of code in the backpack
  - A home-cooked Oanda-like API to connect to the server with endpoints, mix-ins for the calls, separation between the public API and private API and even a Streamer class
  - Taking into account Kraken has only a textual reference of the REST AP, some serious hours of work had been done
  - And the Store pattern was used creating a `KrakenStore`
- The 1st small hurdle … clean integration of the package in backtrader without integrating into the sources. The easiest part actually even if it may seem the more abstract
  - A package (btkraken) with its corresponding `__init__` was created and auto-injection of the `KrakenStore` into backtrader added

```
class MetaSingleton(MetaParams):
    '''Metaclass to make a metaclassed class a singleton'''
    def __init__(cls, name, bases, dct):
        super(MetaSingleton, cls).__init__(name, bases, dct)
        cls._singleton = None
        setattr(bt.stores, name, cls)  # HERE IS THE AUTOINJECTION
```

  - The metaclass of the KrakenStore autoinjects the class during initialization into `bt.stores`
  - With that … one can later simply do
    - mystore = bt.stores.KrakenStore(**the_arguments)
- Why the singleton? To make sure only 1 KrakenStore is created and centralizes all comms

Kraken – CryptoCurrencies – Data Feed 1

- After the autoinjection of the store, the data feed can also autoinject itself in the store

```
class MetaKrakenData(DataBase.__class__):
   def __init__(cls, name, bases, dct):
        '''Class has already been created ... register'''
        # Initialize the class
        super(MetaKrakenData, cls).__init__(name, bases, dct)

        # Register with the store
        bt.stores.KrakenStore.DataCls = cls
```

- Having an integrated store and the data feed, a simple script to fetch the data and resample it to 1-minute was fired
- Things were still not working 100% so and hands on session started with a 1st discussion, printing or debugging?
- Backtrader makes no use of logging but extensive output to the console is used with each new feature and especially with the samples
- After some showing of printing … the team settled for `print` as the best tool for the job … simply because the Kraken documentation tells you what you get with terms like “array of …”
- Even knowing it is JSON … human beings still need to see what they have been given

Kraken – CryptoCurrencies – Data Feed 2

- And printing saved the world. The data feed implementation was so complete, that the only thing missing was actually was … “error” as a key is always present in the messages and not when an “error” happens
- Incredibly … when an error happens the message stops being a dictionary and is a list … where obviously there can be nothing like “error” as a key
- In a matter of minutes and with that corrected ticks were being received
  - The prices were put into the framework
  - And … the time had to be taken from the clock … because the ticks come with no timestamp … hmmmm!
- But an architectural problem was noticed … and it was not in the backtrader architecture, luckily
- Kraken has two things, one in place and one missing for data feeds
  - Rate limiting (after some calculations and with the existing tiers … the limit would be around 30 calls per minute)
  - No price streaming over plain-old chunked-transfer HTTP or websockets or … which means that ticks have to be polled
- And polling “ticks” with a maximum frequency of less than 30 times a minute doesn’t feel like getting ticks (and some calls have to be saved for account updates, order status)

Kraken – CryptoCurrencies – Data Feed 3

- With that in mind, the team decided to turn things around, using the fact that historical download was already in place (smallest timeframe is 1 minute) - change the paradigm to fetch bars rather than ticks
- The OHLC bars are delivered complete with the time boundary but this conflicts with the resampler which uses the realtime clock to avoid delivering a resampled bar too late
- Poll (with a buffer) as close as possible to the boundary, push the resampler a bit into the future (tzoffset) to have the most complete information and not be too late (take into account the drift … to avoid polling going over the boundary)
- And in a matter of minutes … the 1-minutes were flowing into the platform
- Success!!! Ok. Some small hurdles had to be overcome, to synchronize the fetching of data with the latest delivery of Kraken, based on the actual requested timeframe.
- Of course the data feed will still need some polishing, but that’s only a matter of putting some extra time in it.

Kraken – CryptoCurrencies – Data Feed 4

- But the tick-historical approach confirmed what has been answered in the Community several times about a definitive guideline for the integration of an external data feed: “Each provider has its own way”.

- In most cases the basic components are all there: historical download and tick notification, but as seen here … it’ not always the case. It needs

Kraken – CryptoCurrencies – The Broker

- With Rate Limiting in mind as an initial barrier, some other things are particular about Kraken as a broker
- Order status is NOT notified … polling shall it be
- Account status is NOT notified … polling shall it be
- It has a call to fetch open positions, but it doesn't do what other brokers do … you buy some XRPXBT (Ripple – Bitcoin) and … there is NO position, but the Oanda code that was the basis for the team … already calculates the positions
- Luckily and having met the polling problem with the Data Feed and with the Streamer in place that the team brought to the table it was easy
- At the end of the Day2 … the broker could:
  - Send Orders (at least Market and Limit) which were executed
  - Understand the order status changes from the broker

The End