Track Port

[1. Legacy 2](#_Toc54352528)

[1.1. Database 2](#_Toc54352529)

[1.1.1. Table transaction\_list 2](#_Toc54352530)

[1.1.1.1. Describe 2](#_Toc54352531)

[1.1.1.2. Additional Info 2](#_Toc54352532)

[1.1.2. Table port\_history 3](#_Toc54352533)

[1.1.2.1. Describe 3](#_Toc54352534)

[1.1.2.2. Additional Info 3](#_Toc54352535)

[1.1.3. Table finance\_quote 3](#_Toc54352536)

[1.1.3.1. Describe 3](#_Toc54352537)

[1.1.3.2. Additional Info 4](#_Toc54352538)

[1.2. Scripts 4](#_Toc54352539)

[1.1.4. Perl quote\_query 4](#_Toc54352540)

[1.1.4.1. The Main Loop 4](#_Toc54352541)

[1.1.4.2. Function db\_get\_symbols 4](#_Toc54352542)

[1.1.4.3. Function get\_quoteoption\_data 5](#_Toc54352543)

[1.1.4.4. Function get\_quote\_data1 5](#_Toc54352544)

[1.1.5. Perl put\_db\_quotes 6](#_Toc54352545)

[1.1.5.1. The Main Loop 6](#_Toc54352546)

[1.1.5.2. Function db\_parse\_transactions 6](#_Toc54352547)

[1.1.5.3. Function build\_fq\_hash 7](#_Toc54352548)

[1.1.5.4. Function create\_transaction\_report 7](#_Toc54352549)

[1.1.5.5. DB Operations on transaction\_report 8](#_Toc54352550)

[1.1.5.6. DB Operations on port\_param 8](#_Toc54352551)

[1.1.5.7. DB Operations on port\_history 8](#_Toc54352552)

[1.1.6. Perl port\_edit.cgi 8](#_Toc54352553)

[1.1.6.1. General Functions 9](#_Toc54352554)

[1.1.6.2. Form Functions 9](#_Toc54352555)

[1.1.6.3. Submit Functions 10](#_Toc54352556)

[1.1.7. Python pull\_transaction\_report.py 11](#_Toc54352557)

[1.1.8. Perl port\_track.cgi 12](#_Toc54352558)

# Legacy

The track\_port system has existed in various forms over a number of years. Here I am documenting how track\_port has been working since around 2006. The core of the system is a mysql database and a number of scripts used to maintain/query the database. The details of each of these components will be outlined and described in the following sections, but here is a short summary of each component.

The database consists of a number of tables, but there are three tables that are most important. The table transaction\_list contains information about every position (symbol, open info, close info, etc.). With this information, portfolio tracking is possible. The table port\_history keeps track of portfolio names and their historical values and cash positions. Finally, the table finance\_quote contains daily quote data for all tracked symbols.

There are three main scripts that are used to update/query the database. The script quote\_query is a perl script that runs during the market open and repeatedly looks up quote info for all symbols. It then stores this info in the database finance\_quote table. The script port\_edit.cgi is a perl script used to enter/modify information in the database transaction\_list table. It is accessed using a web query. IOW, this is how positions are opened and closed (as well as providing some modify capability). Finally, the script pull\_transaction\_report.py is a python script used to show tabular portfolio info using a web query.

In addition, the script port\_track.cgi is used to effectively generate a pull\_transaction\_report URL. Essentially, you can multiply select fileportnames and specify a few URL GET parameters. The beauty of this script is that it queries the database for all files as well as ports within each file.

## Database

### Table transaction\_list

#### Describe

mysql> describe transaction\_list;

+--------------+------------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+--------------+------------------+------+-----+---------+----------------+

| id | int(10) unsigned | NO | PRI | NULL | auto\_increment |

| fileportname | varchar(256) | YES | | | |

| symbol | varchar(32) | YES | | | |

| sector | varchar(32) | YES | | | |

| position | varchar(16) | YES | | | |

| descriptor | varchar(16) | YES | | | |

| shares | decimal(14,4) | YES | | 0.0000 | |

| open\_price | decimal(14,4) | YES | | 0.0000 | |

| open\_date | date | YES | | NULL | |

| closed | tinyint(1) | YES | | 0 | |

| close\_price | decimal(14,4) | YES | | 0.0000 | |

| close\_date | date | YES | | NULL | |

| expiration | date | YES | | NULL | |

| strike | decimal(14,4) | YES | | 0.0000 | |

+--------------+------------------+------+-----+---------+----------------+

14 rows in set (0.00 sec)

#### Additional Info

The field position can be “cash” or “long”. The former is used for deposits, withdrawals, dividends, adjustments, etc. Essentially, anything that involves a dollar amount as opposed to an actual position. The latter is used for a security position. The original intent was that this could be “long” or “short” depending on how the position was opened. However, short positions are denoted with negative share counts.

The field descriptor is used to describe the type of position for long positions, ie. stock, call or put. For cash positions, it can be “initial” or “intermediate”. The former will set the date when the portfolio was created.

All the other fields are self-explanatory.

One further note regarding the field sector, dividends are marked as such by setting sector to “dividend”. By doing so, it is possible to include dividends as part of a position’s overall return.

### Table port\_param

#### Describe

mysql> describe port\_param;

+----------------+---------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+----------------+---------------+------+-----+---------+----------------+

| id | int unsigned | NO | PRI | NULL | auto\_increment |

| fileportname | varchar(256) | YES | | | |

| cash | decimal(14,4) | YES | | 0.0000 | |

| total | decimal(14,4) | YES | | 0.0000 | |

| pct\_daygain | decimal(14,4) | YES | | 0.0000 | |

| daygain | decimal(14,4) | YES | | 0.0000 | |

| invested\_total | decimal(14,4) | YES | | 0.0000 | |

| pct\_gain | decimal(14,4) | YES | | 0.0000 | |

| gain | decimal(14,4) | YES | | 0.0000 | |

| pct\_invested | decimal(14,4) | YES | | 0.0000 | |

| basis | decimal(14,4) | YES | | 0.0000 | |

| portnum | int unsigned | YES | | 0 | |

+----------------+---------------+------+-----+---------+----------------+

12 rows in set (0.01 sec)

#### Additional Info

Before the pull\_transaction\_report script was rewritten, this used to be the storage place for summary information about each port. Now the information in the summary table of the resulting pull\_transaction\_report page is calculated on the fly.

I think there are some other places where this table is used to source a list of all tracked files/ports?

Also, the cash and/or total values may be used somewhere?

### Table port\_history

#### Describe

mysql> describe port\_history;

+--------------+------------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+--------------+------------------+------+-----+---------+----------------+

| id | int(10) unsigned | NO | PRI | NULL | auto\_increment |

| date | date | YES | | NULL | |

| fileportname | varchar(256) | YES | | | |

| total | decimal(14,4) | YES | | 0.0000 | |

| cash | decimal(14,4) | YES | | 0.0000 | |

+--------------+------------------+------+-----+---------+----------------+

5 rows in set (0.00 sec)

#### Additional Info

Every day the market is open, a new row is created for each port with that date. The total is the total value of the portfolio (including cash). The cash is the amount of cash in the port on that date.

### Table finance\_quote

#### Describe

mysql> describe finance\_quote;

+------------+------------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+------------+------------------+------+-----+---------+-------+

| symbol | varchar(32) | NO | PRI | | |

| name | varchar(32) | YES | | NULL | |

| last | decimal(14,4) | YES | | 0.0000 | |

| high | decimal(14,4) | YES | | 0.0000 | |

| low | decimal(14,4) | YES | | 0.0000 | |

| date | date | YES | | NULL | |

| time | time | YES | | NULL | |

| net | decimal(14,4) | YES | | 0.0000 | |

| p\_change | decimal(6,2) | YES | | 0.00 | |

| volume | int(10) unsigned | YES | | 0 | |

| avg\_vol | int(10) unsigned | YES | | 0 | |

| bid | decimal(14,4) | YES | | 0.0000 | |

| ask | decimal(14,4) | YES | | 0.0000 | |

| close | decimal(14,4) | YES | | 0.0000 | |

| open | decimal(14,4) | YES | | 0.0000 | |

| day\_range | varchar(64) | YES | | NULL | |

| year\_range | varchar(64) | YES | | NULL | |

| eps | decimal(14,4) | YES | | 0.0000 | |

| pe | decimal(14,4) | YES | | 0.0000 | |

| div\_date | date | YES | | NULL | |

| dividend | decimal(14,4) | YES | | 0.0000 | |

| div\_yield | decimal(14,4) | YES | | 0.0000 | |

| cap | decimal(20,4) | YES | | NULL | |

| ex\_div | date | YES | | NULL | |

| nav | decimal(14,4) | YES | | 0.0000 | |

| yield | decimal(14,4) | YES | | 0.0000 | |

| exchange | varchar(32) | YES | | NULL | |

| success | tinyint(1) | YES | | 0 | |

| errormsg | varchar(40) | YES | | NULL | |

| method | varchar(32) | YES | | NULL | |

+------------+------------------+------+-----+---------+-------+

30 rows in set (0.00 sec)

#### Additional Info

Most of this data is no longer captured as quote services have evolved over the years.

## Scripts

### Perl quote\_query

#### The Main Loop

Call db\_get\_symbols to get two lists (list\_symbols and list\_options).

Initialize finance\_quotes hash to be empty at the start of the loop.

If we have anything in list\_options, call get\_quoteoption\_data() to start filling in finance\_quotes.

Call get\_quote\_data1() to fill in more finance\_quotes entries.

There is some debug code that saves finance\_quotes to a file.

At this point we are ready to start db operations:

1. Lock finance\_quote table.
2. Delete finance\_quote table data (on first loop iteration).
3. Loop over list\_symbols and replace entries in finance\_quote table.
4. Loop over list\_options and replace entries in finance\_quote table.
5. Perform finance\_quote table commit.
6. Unlock finance\_quote table.

#### Function db\_get\_symbols

Essentially we query transaction\_list to create a list of symbols and a list of options. The symbol query has the following criteria:

* position in (‘long’, ‘short’)
* descriptor = ‘stock’
* not closed
* symbol <> ‘^DJI’

We use a UNION to include symbols from ticker\_symbols table (again symbol <> ‘^DJI’).

One special case to consider. We strip out any symbols that are 5 characters where last character is ‘X’ if fetch\_mf is False.

Building the list of options uses a query that selects:

* position in (‘long’, ‘short’)
* not closed
* descriptor in (‘call’, ‘put’)

Then we have to build each option as a symbol using symbol, descriptor, strike, and expiration.

TODO: put an example option symbol here.

#### Function get\_quoteoption\_data

The API is simple, pass in the list of option symbols and a pointer to the finance\_quote hash, fill in the hash with quote data for each option.

This should be a total re-write.

#### Function get\_quote\_data1

Here is what I have in the function header:

# Fetches are done using finance::quote fetch method. The difference between this

# function and get\_quote\_data is that this method works better when finance::quote

# returns a mix of good and bad data on a symbol basis. Empirically, here is what I

# observed. Assume we are fetching data for 500 symbols:

# o After the first fetch, anywhere from 150-300 symbols would have bad data.

# o Subsequent fetches would get a mix of good and bad data, but not the same symbols

# each time.

# o Some symbols seemed resistant to getting good fetches.

#

# Given that, here is the proposed algorithm:

# 1. Attempt a fetch of all symbols.

# 2. Loop through each symbol and determine if fetch was good or bad.

# a. If bad, delete that symbol from the fetch hash.

# b. Save the symbol for the next iteration.

# 3. Merge the resulting fetch hash into the accumulate fetch hash.

# 3. Delete bad symbols from the original list to be fetched and goto 1.

# 4. Repeat the above loop, N times or exit the loop if there are no bad symbols.

# 5. If there are still bad symbols, use aq method to fetch and fill accumulate

# fetch hash.

#

# Three parameters are passed:

# p\_finance\_quotes -- Pointer to the accumulate fetch hash (assume it was emptied before the call).

# p\_list\_symbols -- Pointer to list of symbols to fetch (NOTE: ^DJI is added to this list after aq fetch).

# p\_hash\_put\_stats -- Pointer to put\_stats hash.

This should probably be a total re-write.

### Perl put\_db\_quotes

#### The Main Loop

Call db\_parse\_transactions(). More details below, but this will parse transaction\_list and port\_param tables.

Call build\_fq\_hash(). More details below, but this will create a hash of finance\_quote data parsed from finance\_quote table.

Tease out $quote\_date from finance\_quote data for ^GSPC.

Call create\_transaction\_report(). More details below, but this will update port\_param table and transaction\_report table. The latter is not really used anymore.

After calling all of these functions, there are a series of DB operations performed. Essentially, there are 3 DB operations that touch the tables transaction\_report, port\_param, and port\_history:

1. drop/insert/commit transaction\_report
2. truncate/insert/commit port\_param
3. delete\_rows/insert/commit port\_history

#### Function db\_parse\_transactions

This function creates 3 data structures. Each of the structures is passed in as a pointer to the function and the functions modifies them in place.

@parsed\_transactions is a list of hashes containing all transaction\_list data (including options). It is initialized to an empty hash. There are three queries made to transaction\_list (one is specific to esop descriptors that are no longer used). We query transaction\_list for regular (long, short) positions:

* position in (‘long’, ‘short’)
* descriptor == ‘stock’
* not closed

These rows are fetched and we fill in the hash with file, port, symbol, label, sector, date (open), purchase, qty, id.

Then we query transaction\_list for options positions:

* position in (‘long’, ‘short’)
* descriptor in (‘call’, ‘put’)
* not closed

These rows are fetched and we fill in parsed\_transactions hash with file, port, sector, date (open\_date), purchase (open\_price), qty, id. We also create the option symbol using descriptor, expiration and strike information.

%port\_params is a hash containing data from the port\_param table.

@list\_cashonly\_ports is a list of ports that are cash-only (no open positions).

#### Function build\_fq\_hash

This is actually buried inside a watchdog while loop such that it looks at finance\_quote data and simply delays 60 seconds if it doesn’t find anything valid for ^GSPC.

The function has two parameters, each is returned from the function. @list\_fq\_fields is a list of fields available in the finance\_quote table. %hash\_fq is the hash containing the actual finance\_quote table data. It is a two level hash where the first index is the symbol and the second index is the field name.

Querying the database for the fields is done using:

DESCRIBE finance\_quote;

Querying the finance\_quote data is done using:

SELECT \* FROM finance\_quote ORDER BY symbol;

#### Function create\_transaction\_report

This is just a straight function call after initializing the list @transaction\_report. The function has 5 parameters, all of which are passed as references.

Here is the function header comment pulled from the code:

# create\_transaction\_report

# Input parameters:

# p\_parsed\_transactions -- Pointer to the parsed transactions read from input file.

# p\_list\_cashonly\_ports -- Pointer to the list of cash-only ports.

# p\_port\_params -- Pointer to hash containing port parameters.

# p\_hash\_fq -- Pointer to hash finance quote info.

# Output parameters:

# p\_transaction\_report -- Pointer to list of hashes containing transaction report data.

#

# Description

# There are two passes through the parsed\_transactions/options list. The first pass has

# two objectives:

# 1) Total each portfolio (according to fileportname) and store in port\_params hash.

# 2) As portfolios are built, create a mirror image portfolio that combines transactions

# that involve the same symbol. These are appended to the parsed\_transactions list

# as needed and a uniquified fileportname is created for each by appending '\_combined'

# to the port name.

# The second pass through the parsed\_transactions/options list has a single objective:

# 1) Create transaction\_report data for each transaction/option in the list.

This description is accurate and fairly detailed. The key there is the two-pass algorithm where the uniquified “combined” ports are created.

The interesting thing is that most of the work done by this function is not really needed anymore. Instead that work is done when pull\_transaction\_report URL is called.

#### DB Operations on transaction\_report

Lock the table ( LOCK TABLES transaction\_report WRITE; ).

Delete all the current rows ( DELETE FROM transaction\_report; ).

Insert new rows (execute\_query\_transaction\_report()). I don’t want to get into these details, they will be different in python. But effectively we build an INSERT INTO transaction\_report …; query and execute it.

Commit the changes.

Unlock the table ( UNLOCK TABLES; ).

#### DB Operations on port\_param

Truncate the port\_param table ( TRUNCATE TABLE port\_param; ).

Insert new rows (execute\_query\_port\_param()).

Commit the changes.

#### DB Operations on port\_history

Delete rows from port\_history (only for the current date).

Insert new rows (execute\_query\_port\_history()).

Commit the changes.

### Perl port\_edit.cgi

This script lives in the scgi-bin directory (which means a login/password is required to execute it). It is mainly used to create, edit and close positions. But when called without any arguments, it has present 3 main actions.

The first action presents all files and their ports with radio button selectors. This is how you can edit any particular portfolio. NOTE: this is usually invoked directly from the pull\_transaction\_report page via a button.

The second action has edit fields that allow you to create a totally new portfolio.

The third action is about entering a stock split. More details on this below, but it does not modify the database. Rather it shows the transactions from transaction\_list where the split should be applied.

The main body of the script does the following:

1. Untaint the parameters.
2. Query for transaction\_list fields.
3. Generate the page <head> using inline print statements.
4. Use if-then-else tree to call various form functions (described below) to create the page body. All of these function names end with “\_form”.
5. Close out the page <body> and <html> tags.

In addition to all of the “\_form” functions, there are also function names that begin with “submit\_” that handle the button action to which they are assigned. These are described below.

#### General Functions

##### Function untaint\_params()

I seem to remember this was something that needed to be done for cgi. Essentially it takes the GET params passed in via URL and builds %hash\_params to store them.

##### Function calc\_current\_cash()

Given a specific fileportname, this function calculates the current cash position using several database queries.

SELECT open\_price FROM transaction\_list WHERE ((position = ‘cash’) && (descriptor = ‘initial’) && (fileportname = ‘$fpn’))

SELECT shares,open\_price FROM transaction\_list WHERE ((position = 'long') && (descriptor in ('stock', 'call', 'put')) && (NOT closed) && (fileportname = '$fpn'))

SELECT open\_price FROM transaction\_list WHERE ((position = 'cash') && (descriptor = 'intermediate') && (fileportname = '$fpn'))

SELECT shares,open\_price,close\_price FROM transaction\_list WHERE ((position = 'long') && (descriptor in ('stock', 'call', 'put')) && (closed) && (fileportname = '$fpn'))

The calculated cash position is returned.

The function is called from within new\_cash\_transaction\_form().

#### Form Functions

##### Function default\_form()

This is the how the top-level, 3-action form is generated. Can be called without arguments where it queries the database for all files and their ports. Or it can be called with a file argument to list only the ports in that file.

##### Function show\_transactions\_form()

Given a fileportname, this displays the main editing page for transactions. Open positions (transactions from transaction\_list) are displayed in one table on the left with CLOSE, EDIT, and DELETE buttons. All cash transactions are displayed in a table on the right with EDIT and DELETE buttons.

There are also buttons at the top of each table for creating new transaction\_list entries. There are buttons for Open Position, Add Cash, and Final Cash.

##### Function edit\_transaction\_by\_id\_form()

Given an id, query the transaction\_list table for the corresponding transaction. Then generate a HTML form that allows most of the transaction fields to be modified when submitted.

##### Function edit\_cash\_by\_id\_form()

Given an id, query the transaction\_list table for the corresponding cash transaction. Then generate a HTML form that allows most of the transaction fields to be modified when submitted.

##### Function close\_transaction\_by\_id\_form()

Given an id, query the transaction list table for the corresponding transaction. Then generate a HTML form that can be used to indicate how a position is to be closed. By default, the number of open shares is filled in, but can be changed to match the actual number of shares being closed.

##### Function delete\_transaction\_by\_id\_form()

Given an id, query the transaction list table for the corresponding transaction. The generated HTML form has no editable fields. The only option is a Delete button that will delete the transaction when submitted.

##### Function new\_transaction\_form()

This is how new position transactions are created. It gets called when the Open Position button is clicked from the show\_transactions page. The generated HTML has edit boxes for each of the fields in transaction\_list. The fields are customized based on whether this is a stock or call/put option. The Insert button on the generated page will perform the submit action.

##### Function new\_cash\_transaction\_form()

This is how new cash transactions are created. It gets called when the Add Cash or Final Cash button is clicked from the show\_transactions page. The generated HTML has edit boxes for a cash transaction. The only difference, when using the Final Cash form, an intermediate cash transaction is calculated to result in the final cash amount. The Insert button on the generated page will perform the submit action.

#### Submit Functions

##### Function submit\_new\_port()

Effectively, this will generate an INSERT transaction to the database.

INSERT INTO transaction\_list SET fileportname=’%s’,position=’cash’,descriptor=’initial’,open\_price=’%s’;

##### Function submit\_split()

This generates a SELECT transaction to the database to grab any long transactions for the corresponding symbol.

SELECT \* FROM transaction\_list WHERE ((position = ‘long’) && (symbol = ‘%s’)) order by open\_date;

The returned transactions are simply shown in a table, but nothing is actually changed in the database.

##### Function submit\_edit\_transaction\_by\_id()

This generates an UPDATE transaction to the database after determining if any of the fields have been modified in the form.

UPDATE transaction\_list SET <set-parameters> WHERE (id = ‘$id’);

##### Function submit\_edit\_cash\_by\_id()

Similar to submit\_edit\_transaction\_by\_id(), except for a cash transaction.

##### Function submit\_close\_transaction\_by\_id()

The idea here is to generate an UPDATE transaction to the database to close the transaction with closed, close\_price and close\_date. There is one additional caveat, when the number of shares requested to be closed is less than the number of shares in the position. In that case, we have to generate an INSERT transaction to the database to create a new position transaction for the difference in the number of shares in the original position and the number of shared requested to be closed. Also, when the original position is then updated, the number of shares will be the requested number of shares to close.

##### Function submit\_delete\_transaction\_by\_id()

Generate a DELETE transaction to the database, where the selected transaction (identified by id) is deleted. After that action is performed, the show\_transactions page is returned.

##### Function submit\_new\_transaction()

This function is used to generate an INSERT transaction to the database. But there are a number of conditional paths that increase the complexity. From the comments in the code:

# There are four fields that are semi-optional: shares, open\_price, net\_total, commission.

# If shares is blank, then it is calculated using net\_total and open\_price (commission is ignored).

# If open\_price is blank, then it is calculated using net\_total, shares (and optionally commission).

# If net\_total is blank, then open\_price may be recalculated after subtracting commission.

# If commission is the only thing blank, then ignore net\_total.

There query is built as:  
INSERT INTO transaction\_list SET <set-parameters>;

##### Function submit\_new\_cash\_transaction()

Similar to submit\_new\_transaction(), however there is no need to calculate any of the fields.

### Python pull\_transaction\_report.py

TBD

### Perl port\_chart.cgi

Creates portfolio charts and some statistics.

The structure of the script is as follows:

1. Connect to the database.
2. Perform some date manipulations.
3. Loop over each specified port and pull dates and totals from the port\_history table.
   1. Data is stored in @data\_totals.
   2. Index 0 is the list of dates.
   3. Index 1 is the list of totals for the first port, etc.
4. Do some manipulations to create @data\_pcts (percentage change data points) and @data\_sumdiffs (sum or diff data points).
5. More data manipulations to get max/min/etc for %hash\_stats.
6. Execute code that does html/plotting.

The code that performs html generation and plotting is a little tricky. When the page URL is executed, we run through the script with $plot\_param set to False. There are whole sections of code that are skipped or executed depending on $plot\_param.

The first time through with $plot\_param set to False, we call all of the html\_\* functions. The function html\_body() in particular is called up to 3 times. The first time for ‘totals’, second time for ‘pcts’ and third time for ‘sumdiffs’. What does this mean? Within html\_body, we use the first argument to build a duplicate cgi command with a plot argument. By doing so we execute the script again, but this time $plot\_param is (for example) ‘totals’. This time the code skips the html\_\* calls and instead calls generate\_chart() which returns a png of the requested chart. That png is referenced in the generated html such that the browser displays the chart in place of the reference.

In retrospect, this was a clever implementation, but a bit on the hacky, clunky side. Definitely room for improvement using bokeh.

### Perl port\_track.cgi

This script lives in cgi-bin, so no login/password is required. When executed without arguments it presents a page where the user can (multiply) select portfolios for viewing with pull\_transaction\_report. Effectively, it is used to generate the pull\_transaction\_report.py GET URLs.

It looks up all files and their ports and displays each port in its file column with a multiply-selectable form select box. Below that are some radio buttons that can control how the ports are displayed in pull\_transaction\_report.py.

### Python db\_convert.py

This is simply a helper script that is used to create a sqlite3 version of selected track\_port mysql database tables. It can only be run on the server and can only be run with python3.

There are no arguments required. It creates a file called track\_port.db that can be copied to another windows machine for testing python and sqlalchemy.

# Python

The goal here is to transition all of the track\_port infrastructure to python. At this point, I already have pull\_transaction\_report.py in place of pull\_transaction\_report.cgi. And in doing so, I got rid of the necessity of some of the most burdensome and hacky code (see put\_db\_quotes). For example, the legacy system used to create a separate copy of every portfolio where the positions were “combined”. Now, that is done with pull\_transaction\_report.py script, so there is no need for the “\_combined” copy of each port in the database.

The strategy I will use for transitioning the system to python will be as follows:

* Keep the database implementation intact even though the perl library Finance::Quote drove much of that infrastructure.
* Create a python version of quote\_query with some differences.
  + Get rid of the time looping constructs in favor of running it automatically with cron.
  + Do away with time-related arguments.
  + Add a database argument, to allow comparisons with the existing database.
* Create a python version of port\_chart.cgi.
  + Use bokeh for plotting.
* Create a python version of port\_edit.cgi.
  + Allow editing any transaction\_list row (whether open or closed).
  + Closing open positions should be easier. Ideally, you should enter a symbol and a number of shares. Then you provide either a price per share or a net total. Finally, you need a way of specifying lots.
* Create a python version of port\_track.cgi.
  + Allow creating new portfolios.
  + Tools for managing all portfolios (ie. stock splits).

## Scripts

### quote\_query.py

#### Purpose

Regularly update the database with stock quotes.

#### Who, When, Where

The script will be called from a cron job on a regular, repeating basis. There may be several different crontab lines that use different arguments, depending on time of day and recurrence.

The normal case would be for it to run throughout the market day on a regular basis. However, if the cron job tries to run it before the previous call is finished, it should detect that case and simply return without doing anything. It should also have the ability to check the database for market holidays and abort if the market is closed.

The script will live in my bin directory on the server.

#### Ideas

##### Limit Symbols By Port

For performance reasons, allow the specification of a port (or multiple ports), and only update symbols contained in those ports. That would allow you call this for some subset of ports/symbols where the update would be fast. Meanwhile, you update the universe of all watched symbols less frequently.

##### Use finviz Wisely

Finviz can be given a list of stock symbols and will return details about each symbol. It does not work with mutual fund symbols, nor index symbols.

My stock plan is to pull out just the stock symbols and use Finviz to capture the quote details using Screener and get\_ticker\_details().

My mutual fund plan is to only pull those twice a day. Once very early in the morning (well before market open, say 6:00am) and once late in the evening (well after market close and after MF tickers are updated, say 9:00pm).

My index plan is to only pull three indices throughout the day (^DJI, ^GSPC, ^IXIC). This will be done by calling in to get\_a\_quote.py individually for each index. Furthermore, any remaining indices that are included in watch ports can be done sporadically (maybe once per hour) throughout the day.

#### Arguments

##### Specify File/Port Names

Examples:

--fileportnames=port:fluffgazer

--fileportnames=serp:blink –fileportnames=serp:ircp

The default when not specified is to include all fileportnames in the database.

This argument can be specified multiple times, each time adding a new fileportname to the mix.

##### Specify File Names

Examples:

--filenames=watch

--filenames=serp –filenames=port

The default when not specified is not to do any fileportname inclusions based solely on filename.

This argument can be specified multiple times, each time adding a new filename to the mix.