# Module `pinkfish`

## Sub-modules

- pinkfish.analysis
- pinkfish.benchmark
- pinkfish.fetch
- pinkfish.indicator
- pinkfish.itable
- pinkfish.pfcalendar
- pinkfish.plot
- pinkfish.portfolio
- pinkfish.statistics
- pinkfish.stock_market_calendar
- pinkfish.trade
- pinkfish.utility

## Variables

### Variable `DEBUG`

bool : True to enable DBG() output.

## Functions

### Function `DBG`

```
def DBG(
    s
)
```

Debug print. Enable by setting pf.DEBUG=True.

# Module `pinkfish.analysis`

Analysis of results.

This module contains some functions that were copied or derived from the book "Trading Evolved" by Andreas F. Clenow. Below is a correspondance I had with the author:

---

Farrell October 25, 2019 at 15:49 Hi Andreas,
I just finished reading the book. Awesome one of a kind! Thanks so much. I also enjoyed your other two. Question: what is the copyright (if any) on the source code you have in the book. I want to incorporate some of it into my open source backtester, Pinkfish. How should I credit your work if no copyright. I could add a comment at the beginning of each derived function or module at a minimum.
## Farrell
Andreas Clenow October 25, 2019 at 17:29 Hi Farrell,
I can be paid in reviews and/or beer. :)
For an open source project, use the code as you see fit. A credit in the comments somewhere would be nice, but I won't sue you if you forget it.
ac

---

## Functions

### Function `holding_period_map`

```
def holding_period_map(
    dbal
)
```

Display holding period returns in a table.

This shows what your annualized return would have been, had you started this strategy at the start of a given year, as shown in the leftmost column, and held it for a certain number of years. Length of returns should be 30 or less, otherwise the output will be jumbled.

Parameters

**dbal : pd.Series** The daily closing balance indexed by date.

Returns

**None**

Examples

```
>>> table = holding_period_map(dbal['close'])
>>> display(HTML(table))
Years    1   2   3   4   5   6   7   8
2013    30  20  13  12  13  10  12  12
2014    11   5   7  10   6  10   9
...
2020     8
```

**Function kelly_criterian**

```
def kelly_criterian(
    stats,
    benchmark_stats=None
)
```

Use this function to help with sizing of leverage.

This function uses ideas based on the Kelly Criterian.

Parameters

**stats : pd.Series** Statistics for the strategy.
**bbenchmark_stats : pd.Series, optimal** Statistics for the benchmark (default is None, which implies that a benchmark is not being used).

Returns

**s : pf.Series** Leverage statistics.

- sharpe_ratio is a measure of risk adjusted return.
- sharpe_ratio_max is the maximum expected sharpe ratio.
- sharpe_ratio_min is the minimum expected sharpe ratio.
- strategy risk is a measure of how risky a trading strategy is, calculated as an annual standard deviation of returns.
- instrument_risk is a measure of how risky an instrument is before any leverage is applied, calculated as an annual standard deviation of returns.
- optimal target risk is equal to the expected sharpe ratio, according to the Kelly criterian. Target risk is the amount of risk you expect to see when trading, calculated as an annual standard deviation of returns.
- half kelly criterian is equal to half the expected sharpe ratio. It uses a conservative version of the Kelly criterian known as half Kelly.
- aggressive leverage is the optimal target risk divided by the instrument risk. This is an aggressive form of the leverage factor, which is the cash value of a position divided by your capital.
- moderate leverage is the leverage factor calculated using half Kelly.

- conservative leverage is the leverage factor calculated using half of the minimum sharpe ratio divided by 2.

**Function `monthly_returns_map`**

```
def monthly_returns_map(
    dbal
)
```

Display per month and per year returns in a table.

Parameters

**`dbal` : pd.Series** The daily closing balance indexed by date.

Returns

**None**

Examples

```
>>> monthly_returns_map(dbal['close'])
Year    Jan     Feb     Mar     Apr     May     Jun     Jul ... Year
1990   -8.5     0.9     2.4    -2.7     9.2    -0.9    -0.5    -8.2
1991    4.2     6.7     2.2     0.0     3.9    -4.8     4.5    26.3
```

**Function `prettier_graphs`**

```
def prettier_graphs(
    dbal,
    benchmark_dbal,
    dbal_label='Strategy',
    benchmark_label='Benchmark',
    points_to_plot=None
)
```

Plot 3 subplots.

The first subplot will show a rebased comparison of the returns to the benchmark returns, recalculated with the same starting value of 1. This will be shown on a semi logarithmic scale. The second subplot will show relative strength of the returns to the benchmark returns, and the third the correlation between the two.

Parameters

**`dbal` : pd.Series** Strategy daily closing balance indexed by date.
**`benchmark_dbal` : pd.Series** Benchmark daily closing balance indexed by date.
**`label` : str, optional** Label to use in graph for strategy (default is 'Strategy').
**`benchmark_label` : str, optional** Label to use in graph for benchmark (default is 'Benchmark').
**`points_to_plot` : int, optional** Define how many points (trading days) we intend to plot (default is None, which implies plot all points or days).

Returns

**None**

Examples

```
>>> prettier_graphs(dbal['close'], benchmark_dbal['close'],
                    points_to_plot=5000)
```

**Function `volatility_graphs`**

```
def volatility_graphs(
    dbals,
    labels,
```

```
        points_to_plot=None
    )
```

Plot volatility graphs.

The first graph is a boxplot showing the differences between 2 or more returns. The second graph shows the volatility plotted for 2 or more returns.

Parameters

**dbals : list of pd.DataFrame** A list of daily closing balances (or daily instrument closing prices) indexed by date.
**labels : list of str** A list of labels.
**points_to_plot : int, optional** Define how many points (trading days) we intend to plot (default is None, which implies plot all points or days).

Returns

**pf.DataFrame** Statistics comparing the dbals.

Examples

```
>>> df = pf.volatility_graph([ts, dbal], ['SPY', 'Strategy'],
                             points_to_plot=5000)
>>> df
```

# Module `pinkfish.benchmark`

Benchmark for comparision to a strategy.

## Classes

### Class `Benchmark`

```
    class Benchmark(
        symbols,
        capital,
        start,
        end,
        dir_name='data',
        use_adj=False,
        use_continuous_calendar=False,
        force_stock_market_calendar=False
    )
```

Portfolio Benchmark for comparison to a strategy.

Initialize instance variables.

#### Parameters

**symbols : str or list of str** The symbol(s) to use in the benchmark.
**capital : int** The amount of money available for trading.
**start : datetime.datetime** The desired start date for the benchmark.
**end : datetime.datetime** The desired end date for the benchmark.
**dir_name : str, optional** The leaf data dir name (default is 'data').
**use_adj : bool, optional** True to adjust prices for dividends and splits (default is False).
**use_continuous_calendar : bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
**force_stock_market_calendar : bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False.

**Attributes**

`symbols` : **list of str** The symbols to use in the benchmark.
`capital` : **int** The amount of money available for trading.
`start` : **datetime.datetime** The desired start date for the benchmark.
`end` : **datetime.datetime** The desired end date for the benchmark.
`dir_name` : **str, optional** The leaf data dir name (default is 'data').
`use_adj` : **bool, optional** True to adjust prices for dividends and splits.
`use_continuous_calendar` : **bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
`force_stock_market_calendar` : **bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False.
`ts` : **pd.DataFrame** The timeseries of the symbol used in backtest.
`tlog` : **pd.DataFrame** The trade log.
`dbal` : **pd.DataFrame** The daily balance.
`stats` : **pd.Series** The statistics for the benchmark.

**Methods**

**Method `run`**

```
def run(
    self
)
```

Run the strategy.

**Class `Strategy`**

```
class Strategy(
    symbols,
    capital,
    start,
    end,
    dir_name='data',
    use_adj=False,
    use_continuous_calendar=False,
    force_stock_market_calendar=False
)
```

Portfolio Benchmark for comparison to a strategy.

Initialize instance variables.

**Parameters**

`symbols` : **str or list of str** The symbol(s) to use in the benchmark.
`capital` : **int** The amount of money available for trading.
`start` : **datetime.datetime** The desired start date for the benchmark.
`end` : **datetime.datetime** The desired end date for the benchmark.
`dir_name` : **str, optional** The leaf data dir name (default is 'data').
`use_adj` : **bool, optional** True to adjust prices for dividends and splits (default is False).
`use_continuous_calendar` : **bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
`force_stock_market_calendar` : **bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous

timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False.

**Attributes**

`symbols` : **list of str** The symbols to use in the benchmark.
`capital` : **int** The amount of money available for trading.
`start` : **datetime.datetime** The desired start date for the benchmark.
`end` : **datetime.datetime** The desired end date for the benchmark.
`dir_name` : **str, optional** The leaf data dir name (default is 'data').
`use_adj` : **bool, optional** True to adjust prices for dividends and splits.
`use_continuous_calendar` : **bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
`force_stock_market_calendar` : **bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False.
`ts` : **pd.DataFrame** The timeseries of the symbol used in backtest.
`tlog` : **pd.DataFrame** The trade log.
`dbal` : **pd.DataFrame** The daily balance.
`stats` : **pd.Series** The statistics for the benchmark.

**Methods**

**Method `run`**

```
def run(
    self
)
```

Run the strategy.

# Module `pinkfish.fetch`

Fetch time series data.

## Functions

**Function `fetch_timeseries`**

```
def fetch_timeseries(
    symbol,
    dir_name='data',
    use_cache=True,
    from_year=None
)
```

Read time series data.

Use cached version if it exists and use_cache is True, otherwise retrive, cache, then read.

Parameters

`symbol` : **str** The symbol for a security.
`dir_name` : **str, optional** The leaf data dir name (default is 'data').
`use_cache` : **bool, optional** True to use data cache. False to retrieve from the internet (default is True).
`from_year` : **int, optional** The start year for timeseries retrieval (default is None, which implies that all the available data is retrieved).

Returns

**pd.DataFrame** The timeseries of a symbol.

**Function** `finalize_timeseries`

```
def finalize_timeseries(
    ts,
    start,
    dropna=False
)
```

Finalize timeseries.

Drop all rows that have nan column values. Set timeseries to begin at start.

Parameters

**`ts` : pd.DataFrame** The timeseries of a symbol.
**`start` : datetime.datetime** The start date for backtest.
**`dropna` : bool, optional** Drop rows that have a NaN value in one of it's columns (default is True).

Returns

**datetime.datetime** The start date.
**pd.DataFrame** The timeseries of a symbol.

**Function** `get_symbol_metadata`

```
def get_symbol_metadata(
    symbols=None,
    dir_name='data',
    from_year=None
)
```

Get symbol metadata for list of symbols.

Filter out any filename prefixed with '___'.

Parameters

**`symbols` : str or list, optional** The symbol(s) for which to get symbol metadata (default is None, which imples get symbol metadata for all symbols).
**`dir_name` : str, optional** The leaf data dir name (default is 'data).
**`from_year` : int, optional** The start year for timeseries retrieval (default is None, which implies that all the available data is retrieved).

Returns

**pd.DataFrame** Each row contains metadata for a symbol.

**Function** `remove_cache_symbols`

```
def remove_cache_symbols(
    symbols=None,
    dir_name='data'
)
```

Remove cached timeseries for list of symbols.

Filter out any symbols prefixed with '___'.

Parameters

**`symbols` : str or list of str, optional** The symbol(s) for which to remove cached timeseries (default is None, which imples remove timeseries for all symbols).
**`dir_name` : str, optional** The leaf data dir name (default is 'data').

Returns

**None**

**Function `select_tradeperiod`**

```
def select_tradeperiod(
    ts,
    start,
    end,
    use_adj=False,
    use_continuous_calendar=False,
    force_stock_market_calendar=False,
    check_fields=['close']
)
```

Select the trade period.

First, remove rows that have zero values in price columns. Then, select a time slice of the data to trade from ts. Back date a year to allow time for long term indicators, e.g. 200sma is become valid.

Parameters

**`ts` : pd.DataFrame** The timeseries of a symbol.
**`start` : datetime.datetime** The desired start date for the strategy.
**`end` : datetime.datetime** The desired end date for the strategy.
**`use_adj` : bool, optional** True to adjust prices for dividends and splits (default is False).
**`use_continuous_calendar` : bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
**`force_stock_market_calendar` : bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False (default is False).
**`check_fields` : list of str, optional `{'high', 'low', 'open',`** 'close', 'adj_close'} Fields to check for for NaN values. If a NaN value is found for one of these fields, that row is dropped (default is ['close']).

Returns

**pd.DataFrame** The timeseries for specified start:end, optionally with prices adjusted.

Notes

You should only set one of use_continuous_calendar=True or force_stock_market_calendar=True for a continuous timeseries. You should set neither of these to True if your timeseries is based on the stock market.

**Function `update_cache_symbols`**

```
def update_cache_symbols(
    symbols=None,
    dir_name='data',
    from_year=None
)
```

Update cached timeseries for list of symbols.

Filter out any filename prefixed with '___'.

Parameters

**`symbols` : str or list, optional** The symbol(s) for which to update cached timeseries (default is None, which imples update timeseries for all symbols).
**`dir_name` : str, optional** The leaf data dir name (default is 'data).
**`from_year` : int, optional** The start year for timeseries retrieval (default is None, which implies that all the available data is retrieved).

Returns

**None**

# Module `pinkfish.indicator`

Custom indicators.

These indicators are meant to supplement the TA-Lib. See: https://ta-lib.org/function.html

## Functions

### Function `ANNUALIZED_RETURNS`

```
def ANNUALIZED_RETURNS(
    ts,
    lookback=5,
    price='close',
    prevday=False
)
```

Calculate the rolling annualized returns.

Parameters

**`ts` : pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.

**`lookback` : float, optional** The number of years to lookback, e.g. 5 years. 1/12 can be used for 1 month. Likewise 3/12 for 3 months, etc… (default is 5).

**`price` : str, optional `{'close', 'open', 'high', 'low'}`** Input_array column to use for price (default is 'close').

**`prevday` : bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's Volatility (default is False).

Returns

**`s` : pd.Series** Series that contains the rolling annualized returns.

Raises

**ValueError** If the lookback is not positive.

Examples

```
>>> annual_returns_1mo = pf.ANNUALIZED_RETURNS(ts, lookback=1/12)
>>> annual_returns_3mo = pf.ANNUALIZED_RETURNS(ts, lookback=3/12)
>>> annual_returns_1yr = pf.ANNUALIZED_RETURNS(ts, lookback=1)
>>> annual_returns_3yr = pf.ANNUALIZED_RETURNS(ts, lookback=3)
>>> annual_returns_5yr = pf.ANNUALIZED_RETURNS(ts, lookback=5)
```

### Function `ANNUALIZED_SHARPE_RATIO`

```
def ANNUALIZED_SHARPE_RATIO(
    ts,
    lookback=5,
    price='close',
    prevday=False,
    risk_free=0
)
```

Calculate the rolling annualized sharpe ratio.

Parameters

**`ts` : pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.

`lookback` : **float, optional** The number of years to lookback, e.g. 5 years. 1/12 can be used for 1 month. Likewise 3/12 for 3 months, etc… (default is 5).

`price` : **str, optional** `{'close', 'open', 'high', 'low'}` Input_array column to use for price (default is 'close').

`prevday` : **bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's Volatility (default is False).

`risk_free` : **float, optional** The risk free rate (default is 0).

Returns

`s` : **pd.Series** Series that contains the rolling annualized sharpe ratio.

Raises

**ValueError** If the lookback is not positive.

Examples

```
>>> sharpe_ratio_1mo = pf.ANNUALIZED_SHARPE_RATIO(ts, lookback=1/12)
>>> sharpe_ratio_3mo = pf.ANNUALIZED_SHARPE_RATIO(ts, lookback=3/12)
>>> sharpe_ratio_1yr = pf.ANNUALIZED_SHARPE_RATIO(ts, lookback=1)
>>> sharpe_ratio_3yr = pf.ANNUALIZED_SHARPE_RATIO(ts, lookback=3)
>>> sharpe_ratio_5yr = pf.ANNUALIZED_SHARPE_RATIO(ts, lookback=5)
```

**Function** `ANNUALIZED_STANDARD_DEVIATION`

```
def ANNUALIZED_STANDARD_DEVIATION(
    ts,
    lookback=3,
    price='close',
    prevday=False
)
```

Calculate the rolling annualized standard deviation.

Parameters

`ts` : **pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.

`lookback` : **float, optional** The number of years to lookback, e.g. 5 years. 1/12 can be used for 1 month. Likewise 3/12 for 3 months, etc… (default is 5).

`price` : **str, optional** `{'close', 'open', 'high', 'low'}` Input_array column to use for price (default is 'close').

`prevday` : **bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's Volatility (default is False).

Returns

`s` : **pd.Series** Series that contains the rolling annualized standard deviation.

Raises

**ValueError** If the lookback is not positive.

Examples

```
>>> std_dev_1mo = pf.ANNUALIZED_STANDARD_DEVIATION(ts,lookback=1/12)
>>> std_dev_3mo = pf.ANNUALIZED_STANDARD_DEVIATION(ts, lookback=3/12)
>>> std_dev_1yr = pf.ANNUALIZED_STANDARD_DEVIATION(ts, lookback=1)
>>> std_dev_3yr = pf.ANNUALIZED_STANDARD_DEVIATION(ts, lookback=3)
>>> std_dev_5yr = pf.ANNUALIZED_STANDARD_DEVIATION(ts, lookback=5)
```

**Function** `CROSSOVER`

```
def CROSSOVER(
    ts,
    timeperiod_fast=50,
```

```
        timeperiod_slow=200,
        func_fast={'name': 'SMA', 'group': 'Overlap Studies', 'display_name': 'Simple Moving Averag
        func_slow={'name': 'SMA', 'group': 'Overlap Studies', 'display_name': 'Simple Moving Averag
        band=0,
        price='close',
        prevday=False
    )
```

This indicator is used to represent regime direction and duration.

For example, an indicator value of 50 means a bull market that has persisted for 50 days, whereas -20 means a bear market that has persisted for 20 days.

More generally, this is a crossover indicator for two moving averages. The indicator is positive when the fast moving average is above the slow moving arverage, and negative when the fast moving average is below the slow moving average.

Parameters

**ts : pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.
**timeperiod_fast : int, optional** The timeperiod for the fast moving average (default is 50).
**timeperiod_slow : int, optional** The timeperiod for the slow moving average (default is 200).
**func_fast : ta_lib.Function, optional** {SMA, DEMA, EMA, KAMA, T3, TEMA, TRIMA, WMA} The talib function for fast moving average (default is SMA). MAMA not compatible.
**func_slow : ta_lib.Function, optional** {SMA, DEMA, EMA, KAMA, T3, TEMA, TRIMA, WMA} The talib function for slow moving average. (default is SMA). MAMA not compatible.
**band : float, {0-100}, optional** Percent band around the slow moving average. (default is 0, which implies no band is used).
**price : str, optional {'close', 'open', 'high', 'low'}** Input_array column to use for price (default is 'close').
**prevday : bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's CrossOver (default is False).

Returns

**s : pd.Series** Series that contains the rolling regime indicator values.

Raises

**TradeCrossOverError** If one of the timeperiods specified is invalid.

Examples

```
>>> ts['regime'] = pf.CROSSOVER(ts, timeperiod_fast=50,
                                timeperiod_slow=200)
```

**Function MOMENTUM**

```
    def MOMENTUM(
        ts,
        lookback=1,
        time_frame='monthly',
        price='close',
        prevday=False
    )
```

This indicator is used to represent momentum is security prices.

Percent price change is used to calculate momentum. Momentum is positive if the price since the lookback period has increased. Likewise, if price has decreased since the lookback period, momentum is negative. Percent change is used to normalize asset prices for comparison.

Parameters

**ts : pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.
**lookback : int, optional** The number of time frames to lookback, e.g. 2 months (default is 1).

**timeframe : str, optional {'monthly', 'daily', 'weekly', 'yearly'}** The unit or timeframe type of lookback (default is 'monthly').

**price : str, optional {'close', 'open', 'high', 'low'}** Input_array column to use for price (default is 'close').

**prevday : bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's Momentum (default is False).

Returns

**s : pd.Series** Series that contains the rolling momentum indicator values.

Raises

**ValueError** If the lookback is not positive or the time_frame is invalid.

Examples

```
>>> ts['mom'] = pf.MOMENTUM(ts, lookback=6, time_frame='monthly')
```

#### Function `VOLATILITY`

```
def VOLATILITY(
    ts,
    lookback=20,
    time_frame='yearly',
    downside=False,
    price='close',
    prevday=False
)
```

This indicator is used to represent volatility in security prices.

Volatility is represented as the standard deviation. Volatility is calculated over the lookback period, then we scale to the time frame. Volatility scales with the square root of time. For example, if the market's daily volatility is 0.5%, then volatility for two days is the square root of 2 times the daily volatility (0.5% * 1.414 = 0.707%). We use the square root of time to scale from daily to weely, monthly, or yearly.

Parameters

**ts : pd.DateFrame** A dataframe with 'open', 'high', 'low', 'close', 'volume'.

**lookback : int, optional** The number of time frames to lookback, e.g. 2 months (default is 1).

**timeframe : str, optional {'yearly', 'daily', 'weekly', 'monthly'}** The unit or timeframe used for scaling. For example, if the lookback is 20 and the timeframe is 'yearly', then we compute the 20 day volatility and scale to 1 year. (default is 'yearly').

**downside : bool, optional** True to calculate the downside volatility (default is False).

**price : str, optional {'close', 'open', 'high', 'low'}** Input_array column to use for price (default is 'close').

**prevday : bool, optional** True will shift the series forward. Unless you are buying on the close, you'll likely want to set this to True. It gives you the previous day's Volatility (default is False).

Returns

**s : pd.Series** A new column that contains the rolling volatility.

Raises

**ValueError** If the lookback is not positive or the time_frame is invalid.

Examples

```
>>> ts['vola'] = pf.VOLATILITY(ts, lookback=20, time_frame='yearly')
```

## Classes

#### Class `IndicatorError`

```
class IndicatorError(
```

```
    *args,
    **kwargs
)
```

Base indicator exception.

#### Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

#### Descendants

- pinkfish.indicator.TradeCrossOverError

#### Class `TradeCrossOverError`

```
class TradeCrossOverError(
    *args,
    **kwargs
)
```

Invalid timeperiod specified.

#### Ancestors (in MRO)

- pinkfish.indicator.IndicatorError
- builtins.Exception
- builtins.BaseException

## Module `pinkfish.itable`

Keep track of styles for cells/headers in PrettyTable.

The MIT License (MIT)

Copyright (c) 2014 Melissa Gymrek mgymrek@mit.edu[1]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### Classes

#### Class `CellStyle`

```
class CellStyle
```

Styles for cells PrettyTable

[1] mailto:mgymrek@mit.edu

### Methods

#### Method `column_format`

```
def column_format(
    self,
    x
)
```

#### Method `copy`

```
def copy(
    self
)
```

#### Method `css`

```
def css(
    self
)
```

#### Method `set`

```
def set(
    self,
    key,
    value
)
```

### Class `PrettyTable`

```
class PrettyTable(
    df,
    tstyle=None,
    header_row=False,
    header_col=True,
    center=False,
    rpt_header=0
)
```

Formatted tables for display in IPython notebooks

df: pandas.DataFrame style: TableStyle header_row: include row headers header_col: include column headers

### Methods

#### Method `copy`

```
def copy(
    self
)
```

#### Method `reset_cell_style`

```
def reset_cell_style(
    self,
    rows=None,
    cols=None
)
```

Reset existing cell style to defaults

**Method `reset_col_header_style`**

```
def reset_col_header_style(
    self,
    indices=None
)
```

Reset col header style to defaults

**Method `reset_corner_style`**

```
def reset_corner_style(
    self
)
```

Reset corner style to defaults

**Method `reset_row_header_style`**

```
def reset_row_header_style(
    self,
    indices=None
)
```

Reset row header style to defaults

**Method `set_cell_style`**

```
def set_cell_style(
    self,
    style=None,
    tuples=None,
    rows=None,
    cols=None,
    format_function=None,
    **kwargs
)
```

Apply cell style to rows and columns specified

**Method `set_col_header_style`**

```
def set_col_header_style(
    self,
    style=None,
    indices=None,
    format_function=None,
    **kwargs
)
```

Apply style to header at specific index If index is None, apply to all headings

**Method `set_corner_style`**

```
def set_corner_style(
    self,
    style=None,
    format_function=None,
    **kwargs
)
```

Apply style to the corner cell

**Method `set_row_header_style`**

```
def set_row_header_style(
    self,
    style=None,
    indices=None,
    format_function=None,
    **kwargs
)
```

Apply style to header at specific index If index is None, apply to all headings

**Method `update_cell_style`**

```
def update_cell_style(
    self,
    rows=None,
    cols=None,
    format_function=None,
    **kwargs
)
```

Update existing cell style

**Method `update_col_header_style`**

```
def update_col_header_style(
    self,
    indices=None,
    format_function=None,
    **kwargs
)
```

Update existing row header tyle

**Method `update_corner_style`**

```
def update_corner_style(
    self,
    format_function=None,
    **kwargs
)
```

Update the corner style

**Method `update_row_header_style`**

```
def update_row_header_style(
    self,
    indices=None,
    format_function=None,
    **kwargs
)
```

Update existing row header tyle

**Class `TableStyle`**

```
class TableStyle(
    theme=None
)
```

Keep track of styles for cells/headers in PrettyTable

# Module `pinkfish.pfcalendar`

Adds calendar columns to a timeseries.

- dotw : int, {0-6}
  Day of the week with Monday=0, Sunday=6.
- dotm : int, {1,2,...}
  Day of the month as 1,2,...
- doty : int, {1,2,...}
  Day of the year as 1,2,...
- month : int, {1-12}
  Month as January=1,...,December=12
- first_dotw : bool
  First trading day of the week.
- last_dotw : bool
  Last trading day of the week.
- first_dotm : bool
  First trading day of the month.
- last_dotm : bool
  Last trading day of the month.
- first_doty : bool
  First trading day of the year.
- last_doty : bool
  Last trading day of the year.

## Functions

### Function `calendar`

```
def calendar(
    ts
)
```

Add calendar columns to a timeseries.

Parameters

**`ts` : pd.DataFrame** The timeseries of a symbol.

Returns

**pd.DataFrame** The timeseries with calendar columns added.

# Module `pinkfish.plot`

Plotting functions.

## Variables

### Variable `default_metrics`

tuple : Default metrics for plot_bar_graph().

The metrics are:

```
'annual_return_rate'
'max_closed_out_drawdown'
'annualized_return_over_max_drawdown'
```

```
'best_month'
'worst_month'
'sharpe_ratio'
'sortino_ratio'
'monthly_std'
'annual_std'
```

## Functions

### Function `optimizer_plot_bar_graph`

```
def optimizer_plot_bar_graph(
    df,
    metric
)
```

Plot Bar Graph of a metric for a set of strategies.

This function is designed to be used in analysis of an optimization of some parameter. First call optimizer_summary() to generate the dataframe required by this function.

Parameters

**`df` : pf.DataFrame** Summary of strategies vs metrics.
**`metric` : str** The metric to be used in the summary.

### Function `plot_bar_graph`

```
def plot_bar_graph(
    stats,
    benchmark_stats=None,
    metrics=('annual_return_rate', 'max_closed_out_drawdown', 'annualized_return_over_max_drawd
    extras=None,
    fname=None
)
```

Plot Bar Graph: Strategy vs Benchmark (optional).

Parameters

**`stats` : pd.Series** Statistics from the strategy.
**`benchmark_stats` : pd.Series, optional** Statistics from the benchmark (default is None, which implies that a benchmark is not being used).
**`metrics` : tuple, optional** The metrics to be plotted (default is default_metrics).
**`extras` : tuple, optional** The additional metrics to be plotted (default is None, which implies no extra metrics should be added).
**`fname` : str or `path-like` or `file-like`, optional** Save the current figure to fname (default is None, which implies to not output the figure to a file).

Returns

**pd.DataFrame** Summary metrics.

### Function `plot_equity_curve`

```
def plot_equity_curve(
    strategy,
    benchmark=None,
    yscale='linear',
    fname=None
)
```

Plot Equity Curve: Strategy and (optionally) Benchmark.

Parameters

**strategy : pd.DataFrame** Daily balance for the strategy.
**benchmark : pd.DataFrame, optional** Daily balance for the benchmark (default is None, which implies that a benchmark is not being used).
**yscale : str, {'linear', 'log', 'symlog', 'logit'}** The axis scale type to apply (default is 'linear')
**fname : str or path-like or file-like, optional** Save the current figure to fname (default is None, which implies to not output the figure to a file).

Returns

**None**

**Function plot_equity_curves**

```
def plot_equity_curves(
    strategies,
    labels=None,
    yscale='linear',
    fname=None
)
```

Plot one or more equity curves on the same plot.

Parameters

**strategies : pd.Series of pd.Dataframe** Container of strategy daily balance for each symbol.
**labels : list of str, optional** List of labels for each strategy (default is None, which implies that strategy.symbol is used as the label.
**yscale : str, {'linear', 'log', 'symlog', 'logit'}** The axis scale type to apply (default is 'linear')
**fname : str or path-like or file-like, optional** Save the current figure to fname (default is None, which implies to not output the figure to a file).

Returns

**None**

**Function plot_trades**

```
def plot_trades(
    strategy,
    benchmark=None,
    yscale='linear',
    fname=None
)
```

Plot Trades.

Benchmark is the equity curve that the trades get plotted on. If not provided, strategy equity curve is used.

Parameters

**strategy : pd.DataFrame** Daily balance for the strategy.
**benchmark : pd.DataFrame, optional** Daily balance for the benchmark.
**yscale : str, {'linear', 'log', 'symlog', 'logit'}** The axis scale type to apply (default is 'linear')
**fname : str or path-like or file-like, optional** Save the current figure to fname (default is None, which implies to not output the figure to a file).

Returns

**None**

# Module `pinkfish.portfolio`

Portfolio backtesting.

## Functions

### Function `technical_indicator`

```
def technical_indicator(
    symbols,
    output_column_suffix,
    input_column_suffix='close'
)
```

Decorator for adding a technical indicator to portfolio symbols.

A new column will be added for each symbol. The name of the new column will be the symbol name, an underscore, and the output_column_suffix. For example, 'SPY_MA30' is the symbol SPY with output_column_suffix equal to MA30.

func is a wrapper for a technical analysis function. The actual technical analysis function could be from ta-lib, pandas, pinkfish indicator, or a custom user function.

'func' must have the positional argument ts and keyword argument input_column. 'ts' is passed in, but input_column (args[1]) is assigned in the wrapper before func is called.

Parameters

**`symbols : list`** The symbols that constitute the portfolio.
**`output_column_suffix : str`** Output column suffix to use for technical indicator.
**`input_column_suffix : str, {'close', 'open', 'high', 'low'}`** Input column suffix to use for price (default is 'close').

Returns

**`decorator : function`** A wrapper that adds technical indicators to portfolio symbols.

Examples

```
>>> # Technical indicator: volatility.
>>> @pf.technical_indicator(symbols, 'vola', 'close')
>>> def _volatility(ts, input_column=None):
...     return pf.VOLATILITY(ts, price=input_column)
>>> ts = _volatility(ts)
```

## Classes

### Class `Portfolio`

```
class Portfolio
```

A portfolio or collection of securities.

#### Methods

- fetch_timeseries()
  Get time series data for symbols.
- add_technical_indicator()
  Add a technical indicator for each symbol in the portfolio.
- calendar()
  Add calendar columns.
- finalize_timeseries()
  Finalize timeseries.

- get_price()
  Return price given row, symbol, and field.
- get_prices()
  Return dict of prices for all symbols given row and fields.
- shares()
  Return number of shares for given symbol in portfolio.
- positions
  Gets the active symbols in portfolio as a list.
- share_percent()
  Return share value of symbol as a percentage of total_funds.
- adjust_percent()
  Adjust symbol to a specified weight (percent) of portfolio.
- print_holdings()
  Print snapshot of portfolio holding and values.
- init_trade_logs()
  Add a trade log for each symbol.
- record_daily_balance()
  Append to daily balance list.
- get_logs()
  Return raw tradelog, tradelog, and daily balance log.
- performance_per_symbol()
  Returns performance per symbol data, also plots performance.
- correlation_map()
  Show correlation map between symbols.

Initialize instance variables.

#### Attributes

**`_l` : list of tuples** The list of daily balance tuples.
**`_ts` : pd.DataFrame** The timeseries of the portfolio.
**`symbols` : list** The symbols that constitute the portfolio.

#### Instance variables

#### Variable `positions`

Return the active symbols in portfolio as a list.

This returns only those symbols that currently have shares allocated to them, either long or short.

Parameters

`None`

Returns

**list of str** The active symbols in portfolio.

#### Methods

**Method** `add_technical_indicator`

```
def add_technical_indicator(
    self,
    ts,
    ta_func,
    ta_param,
    output_column_suffix,
    input_column_suffix='close'
)
```

Add a technical indicator for each symbol in the portfolio.

A new column will be added for each symbol. The name of the new column will be the symbol name, an underscore, and the output_column_suffix. For example, 'SPY_MA30' is the symbol SPY with output_column_suffix equal to MA30.

ta_func is a wrapper for a technical analysis function. The actual technical analysis function could be from ta-lib, pandas, pinkfish indicator, or a custom user function. ta_param is used to pass 1 parameter to ta_func. Other parameters could be passed to the technical indicator within ta_func. If you need to mass more than 1 paramters to ta_func, you could make ta_param a dict.

Parameters

**`ts : pd.DataFrame`** The timeseries of the portfolio.
**`ta_func : function`** A wrapper for a technical analysis function.
**`ta_param : object`** The parameter for ta_func (typically an int).
**`output_column_suffix : str`** Output column suffix to use for technical indicator.
**`input_column_suffix : str, {'close', 'open', 'high', 'low'}`** Input column suffix to use for price (default is 'close').

Returns

**`ts : pd.DataFrame`** Timeseries with new column for technical indicator.

Examples

```
>>> # Add technical indicator: X day high
>>> def period_high(ts, ta_param, input_column):
>>>     return pd.Series(ts[input_column]).rolling(ta_param).max()

>>> ts = portfolio.add_technical_indicator(
>>>     ts, ta_func=_period_high, ta_param=period,
>>>     output_column_suffix='period_high'+str(period),
>>>     input_column_suffix='close')
```

**Method** `adjust_percent`

```
def adjust_percent(
    self,
    date,
    price,
    weight,
    symbol,
    row,
    direction='LONG'
)
```

Adjust symbol to a specified weight (percent) of portfolio.

Parameters

**`date : str`** The current date.
**`price : float`** The current price of the security.
**`weight : float`** The requested weight for the symbol.

`symbol : str` The symbol for a security.
`row : pd.Series` A row of data from the timeseries of the portfolio.
`direction : pf.Direction, optional` The direction of the trade (default is pf.Direction.LONG).

Returns

**int** The number of shares bought or sold.

### Method `adjust_percents`

```
def adjust_percents(
    self,
    date,
    prices,
    weights,
    row,
    directions=None
)
```

Adjust symbols to a specified weight (percent) of portfolio.

This function assumes all positions are LONG. Prices and weights are given for all symbols in the portfolio. The ordering of the prices and weights dicts are unimportant. They are dicts which are indexed by the symbol.

Parameters

`date : str` The current date.
`prices : dict of floats` Dict of key value pair of symbol:price.
`weights : dict of floats` Dict of key value pair of symbol:weight.
`row : pd.Series` A row of data from the timeseries of the portfolio.
`directions : dict of pf.Direction, optional` Dict of key value pair of symbol:direction. The direction of the trades (default is None, which implies that all positions are long).

Returns

`w : dict of floats` Dict of key value pair of symbol:weight.

### Method `calendar`

```
def calendar(
    self,
    ts
)
```

Add calendar columns to a timeseries.

Parameters

`ts : pd.DataFrame` The timeseries of a symbol.

Returns

**pd.DataFrame** The timeseries with calendar columns added.

### Method `correlation_map`

```
def correlation_map(
    self,
    ts,
    method='log',
    days=None
)
```

Show correlation map between symbols.

Parameters

**ts : pd.DataFrame** The timeseries of the portfolio.
**method : str, optional {'price', 'log', 'returns'}** Analysis done based on specified method (default is 'log').
**days : int** How many days to use for correlation (default is None, which implies all days).

Returns

**df : pd.DataFrame** The dataframe contains the correlation data for each symbol in the portfolio.

### Method `fetch_timeseries`

```
def fetch_timeseries(
    self,
    symbols,
    start,
    end,
    fields=['open', 'high', 'low', 'close'],
    dir_name='data',
    use_cache=True,
    use_adj=True,
    use_continuous_calendar=False,
    force_stock_market_calendar=False,
    check_fields=['close']
)
```

Fetch time series data for symbols.

Parameters

**symbols : list** The list of symbols to fetch timeseries.
**start : datetime.datetime** The desired start date for the strategy.
**end : datetime.datetime** The desired end date for the strategy.
**fields : list, optional** The list of fields to use for each symbol (default is ['open', 'high', 'low', 'close']).
**dir_name : str, optional** The leaf data dir name (default is 'data').
**use_cache : bool, optional** True to use data cache. False to retrieve from the internet (default is True).
**use_adj : bool, optional** True to adjust prices for dividends and splits (default is False).
**use_continuous_calendar : bool, optional** True if your timeseries has data for all seven days a week, and you want to backtest trading every day, including weekends. If this value is True, then force_stock_market_calendar is set to False (default is False).
**force_stock_market_calendar : bool, optional** True forces use of stock market calendar on timeseries. Normally, you don't need to do this. This setting is intended to transform a continuous timeseries into a weekday timeseries. If this value is True, then use_continuous_calendar is set to False (default is False).
**check_fields : list of str, optional {'high', 'low', 'open',** 'close', 'adj_close'} Fields to check for for NaN values. If a NaN value is found for one of these fields, that row is dropped (default is ['close']).

Returns

**pd.DataFrame** The timeseries of the symbols.

### Method `finalize_timeseries`

```
def finalize_timeseries(
    self,
    ts,
    start,
    dropna=True
)
```

Finalize timeseries.

Drop all rows that have nan column values. Set timeseries to begin at start.

Parameters

**`ts` : pd.DataFrame** The timeseries of a symbol.
**`start` : datetime.datetime** The start date for backtest.
**`dropna` : bool, optional** Drop rows that have a NaN value in one of it's columns (default is True).

Returns

**datetime.datetime** The start date.
**pd.DataFrame** The timeseries of a symbol.

### Method `get_logs`

```
def get_logs(
    self
)
```

Return raw tradelog, tradelog, and daily balance log.

Parameters

`None`

Returns

**`rlog` : pd.DataFrame** The raw trade log.
**`tlog` : pd.DataFrame** The trade log.
**`dbal` : pd.DataFrame** The daily balance log.

### Method `get_price`

```
def get_price(
    self,
    row,
    symbol,
    field='close'
)
```

Return price given row, symbol, and field.

Parameters

**`row` : pd.Series** The row of data from the timeseries of the portfolio.
**`symbol` : str** The symbol for a security.
**`field` : str, optional `{'close', 'open', 'high', 'low'}`** The price field (default is 'close').

Returns

**`price` : float** The current price.

### Method `get_prices`

```
def get_prices(
    self,
    row,
    fields=['open', 'high', 'low', 'close']
)
```

Return dict of prices for all symbols given row and fields.

Parameters

**`row` : pd.Series** A row of data from the timeseries of the portfolio.
**`fields` : list, optional** The list of fields to use for each symbol (default is ['open', 'high', 'low', 'close']).

Returns

**`d` : dict of floats** The price indexed by symbol and field.

### Method `init_trade_logs`

```
def init_trade_logs(
    self,
    ts
)
```

Add a trade log for each symbol.

Parameters

**`ts` : pd.DataFrame** The timeseries of the portfolio.

Returns

**None**

### Method `performance_per_symbol`

```
def performance_per_symbol(
    self,
    weights
)
```

Returns performance per symbol data, also plots performance.

Parameters

**`weights` : dict of floats** A dictionary of weights with symbol as key.

Returns

**`df` : pd.DataFrame** The dataframe contains performance for each symbol in the portfolio.

### Method `print_holdings`

```
def print_holdings(
    self,
    date,
    row,
    percent=False
)
```

Print snapshot of portfolio holding and values.

Includes all symbols regardless of whether a symbol has shares currently allocated to it.

Parameters

**`date` : str** The current date.
**`row` : pd.Series** A row of data from the timeseries of the portfolio.
**`percent` : bool, optional** Show each holding as a percent instead of shares. (default is False).

Returns

**None**

### Method `record_daily_balance`

```
def record_daily_balance(
    self,
    date,
    row
)
```

Append to daily balance list.

The portfolio version of this function uses closing values for the daily high, low, and close.

Parameters

**`date : str`** The current date.
**`row : pd.Series`** A row of data from the timeseries of the portfolio.

Returns

**None**

#### Method `share_percent`

```
def share_percent(
    self,
    row,
    symbol
)
```

Return share value of symbol as a percentage of total_funds.

Parameters

**`row : pd.Series`** A row of data from the timeseries of the portfolio.
**`symbol : str`** The symbol for a security.

Returns

**float** The share value as a percent.

#### Method `shares`

```
def shares(
    self,
    symbol
)
```

Return number of shares for given symbol in portfolio.

Parameters

**`symbol : str`** The symbol for a security.

Returns

**`tlog.shares : int`** The number of shares for a given symbol.

## Module `pinkfish.statistics`

Calculate trading statistics.

The stats() function returns the following metrics in a pd.Series.

- start : str
  The date when trading begins formatted as YY-MM-DD.
- end : str
  The date when trading ends formatted as YY-MM-DD.
- beginning_balance : int
  The initial capital.
- ending_balance : float
  The ending capital.
- total_net_profit : float
  Total value of all profitable trades minus all losing trades.

- gross_profit : float
  Total value of all profitable trades.
- gross_loss : float
  Total value of all losing trades.
- profit_factor : float
  The Ratio of the total profits from profitable trades divided by the total loses from losing trades. A break-even system has a profit factor of 1.
- return_on_initial_capital : float
  The ratio of gross profit divided by the initial capital and multiplied by 100.
- annual_return_rate : float
  The compound annual growth rate of the strategy.
- trading_period : str
  The trading time frame expressed as years, monthe, and days.
- pct_time_in_market : float
  The percentage of days in which the strategy is not completely holding cash.
- margin : float
  The buying power in dollars divided by the capital. For example, if the margin is 2 and the capital is $10,000, then the buying power is $20,000.
- avg_leverage : float
  Leverage is the total value of securities held plus any cash, divided by the total value of securities held plus cash minus loans. The average leverage is just the average daily leverage over the life of the strategy.
- max_leverage : float
  The maximum daily leverage over the life of the strategy.
- min_leverage : float
  The minimum daily leverage over the life of the strategy.
- total_num_trades : int
  The number of closed trades.
- trades_per_year : float
  The average number of closed trades per year.
- num_winning_trades : int
  The number of profitable trades.
- num_losing_trades : int
  The number of losing trades.
- num_even_trades : int
  The number of break even trades.
- pct_profitable_trades : float
  The number of winning trades divided by the total number of closed trades and multiplied by 100.
- avg_profit_per_trade : float
  The total net profit divided by the total number of closed trades and multiplied by 100.
- avg_profit_per_winning_trade : float
  The gross profit divided by the number of winning trades.
- avg_loss_per_losing_trade : float
  The gross loss divided by the number of losing trades. This quantity is negative.
- ratio_avg_profit_win_loss : float
  The absolute value of the average profit per winning trade divided by the average loss per losing trade.

- largest_profit_winning_trade : float
  The single largest profit for all winning trades.
- largest_loss_losing_trade : float
  The single largest loss for all losing trades.
- num_winning_points : float
  The sum of the increase in points from all winning trades.
- num_losing_points : float
  The sum of the decrease in points from all losing trades. This quantity is negative.
- total_net_points : float
  The mathematical difference between winning points and losing points.
- avg_points : float
  The total net points divided by the total number of trades.
- largest_points_winning_trade : float
  The single largest point increase for all winning trades.
- largest_points_losing_trade : float
  The single largest point decrease for all losing trades.
- avg_pct_gain_per_trade : float
  The average percentage gain for all trades.
- largest_pct_winning_trade : float
  The single largest percent increase for all winning trades.
- largest_pct_losing_trade : float
  The single largest percent decrease for all losing trades.
- expected_shortfall : float
  The expected shortfall is calculated by taking the average of returns in the worst 5% of cases. In other words, it is the average percent loss of the worst 5% of losing trades.
- max_consecutive_winning_trades : int
  The longest winning streak in trades.
- max_consecutive_losing_trades : int
  The longest losing streak in trades.
- avg_bars_winning_trades : float
  On average, how long a winning trade takes in market days.
- avg_bars_losing_trades : float
  On average, how long a losing trade takes in market days.
- max_closed_out_drawdown : float
  Worst peak minus trough balance based on closing prices.
- max_closed_out_drawdown_peak_date : str
  The beginning and peak date of the largest drawdown formatted as YY-MM-DD. The balance hit it's highest point on this date.
- max_closed_out_drawdown_trough_date : str
  The trough date of the largest drawdown. The balance hit it's lowest point on this date.
- max_closed_out_drawdown_recovery_date : str
  The end date of the largest drawdown. The date in which the balance has equaled the peak value again.
- drawdown_loss_period : int
  The number of calendar days from peak to trough.
- drawdown_recovery_period : int
  The number of calendar days from trough to recovery.

- annualized_return_over_max_drawdown : float
  Annual return rate divided by the max drawdown.
- max_intra_day_drawdown : float
  Worst peak minus trough balance based on intraday values.
- avg_yearly_closed_out_drawdown :float
  The average yearly drawdown calculated using every available market year period. In other words, every rollowing window of 252 market days is taken as a different year in the calculation.
- max_yearly_closed_out_drawdown : float
  Worst peak minus trough balance based on closing prices during any 252 market day period.
- avg_monthly_closed_out_drawdown : float
  The average monthly drawdown calculated using every available market month period. In other words, every rollowing window of 20 market days is taken as a different month in the calculation.
- max_monthly_closed_out_drawdown : float
  Worst peak minus trough balance based on closing prices during any 20 market day period.
- avg_weekly_closed_out_drawdown : float
  The average weekly drawdown calculated using every available market week period. In other words, every rollowing window of 5 market days is taken as a different week in the calculation.
- max_weekly_closed_out_drawdown : float
  Worst peak minus trough balance based on closing prices during any 5 market day period.
- avg_yearly_closed_out_runup : float
  The average yearly runup calculated using every available market year period. In other words, every rollowing window of 252 market days is taken as a different year in the calculation.
- max_yearly_closed_out_runup : float
  Best peak minus trough balance based on closing prices during any 252 market day period.
- avg_monthly_closed_out_runup : float
  The average monthly runup calculated using every available market month period. In other words, every rollowing window of 20 market days is taken as a different month in the calculation.
- max_monthly_closed_out_runup : float
  Best peak minus trough balance based on closing prices during any 20 market day period.
- avg_weekly_closed_out_runup : float
  The average weekly runup calculated using every available market week period. In other words, every rollowing window of 5 market days is taken as a different week in the calculation.
- max_weekly_closed_out_runup : float
  Best peak minus trough balance based on closing prices during any 5 market day period.
- pct_profitable_years : float
  The percentage of all years that were profitable. In other words, the percentage of 252 market day periods that were profitable.
- best_year : float
  The percentage increase in balance of the best year.
- worst_year : float
  The percentage decrease in balance of the worst year.
- avg_year : float
  The percentage change per year on average.
- annual_std : float
  The yearly standard deviation over the entire trading period.
- pct_profitable_months : float
  The percentage of all months that were profitable. In other words, the percentage of 20 market day periods that were profitable.

- best_month : float
  The percentage increase in balance of the best month.
- worst_month : float
  The percentage decrease in balance of the worst month.
- avg_month : float
  The percentage change per month on average.
- monthly_std : float
  The monthly standard deviation over the entire trading period.
- pct_profitable_weeks : float
  The percentage of all weeks that were profitable. In other words, the percentage of 5 market day periods that were profitable.
- best_week : float
  The percentage increase in balance of the best week.
- worst_week : float
  The percentage decrease in balance of the worst week.
- avg_week : float
  The percentage change per week on average.
- weekly_std : float
  The weekly standard deviation over the entire trading period.
- pct_profitable_weeks : float
  The percentage of all weeks that were profitable. In other words, the percentage of 5 market day periods that were profitable.
- weekly_std : float
  The weekly standard deviation over the entire trading period.
- pct_profitable_days : float
  The percentage of all days that were profitable.
- best_day : float
  The percentage increase in balance of the best day.
- worst_day : float
  The percentage decrease in balance of the worst day.
- avg_day : float
  The percentage change per day on average.
- daily_std : float
  The daily standard deviation over the entire trading period.
- sharpe_ratio : float
  A measure of risk adjusted return. The ratio is the average return per unit of volatility, i.e. standard deviation.
- sharpe_ratio_max : float
  The maximum expected sharpe ratio. It is the sharpe ratio plus 3 standard deviations of the sharpe ratio. 99.73% of sharpe ratios are theoretically below this value.
- sharpe_ratio_min : float
  The mimimum expected sharpe ratio. It is the sharpe ratio minus 3 standard deviations of the sharpe ratio. 99.73% of sharpe ratios are theoretically above this value.
- sortino_ratio : float
  A variation of the Sharpe ratio that differentiates harmful volatility from overall volatility by using the asset's standard deviation of negative portfolio returns (downside deviation) instead of the total standard deviation.

## Variables

### Variable `ALPHA_BEGIN`

tuple : Use with select_timeseries, beginning data for any timeseries.

### Variable `SP500_BEGIN`

tuple : Use with select_timeseries, date the S&P500 began.

### Variable `TRADING_DAYS_PER_MONTH`

int : The number of trading days per month.

### Variable `TRADING_DAYS_PER_WEEK`

int : The number of trading days per week.

### Variable `TRADING_DAYS_PER_YEAR`

int : The number of trading days per year.

### Variable `currency_metrics`

tuple : Currency metrics for summary().

The metrics are:

```
'beginning_balance'
'ending_balance'
'total_net_profit'
'gross_profit'
'gross_loss'
```

### Variable `default_metrics`

tuple : Default metrics for summary().

The metrics are:

```
'annual_return_rate'
'max_closed_out_drawdown'
'best_month'
'worst_month'
'sharpe_ratio'
'sortino_ratio'
'monthly_std'
'annual_std'
```

## Functions

### Function `currency`

```
def currency(
    amount
)
```

Returns the dollar amount in US currency format.

Parameters

**`amount : float`** The dollar amount to convert.

Returns

**`str`** the dollar amount in US currency format.

### Function `optimizer_summary`

```
def optimizer_summary(
    strategies,
    metrics
)
```

Generate summary dataframe of a set of strategies vs metrics.

This function is designed to be used in analysis of an optimization of some parameter. stats() must be called for each strategy before calling this function.

Parameters

**`strategies` : pd.Series** Series of strategy objects that have the stats() attribute.
**`metrics` : tuple** The metrics to be used in the summary.

Returns

**`df` : pf.DataFrame** Summary of strategies vs metrics.

### Function `select_trading_days`

```
def select_trading_days(
    use_stock_market_calendar
)
```

Select between continuous and standard stock market days.

Set use_stock_market_calendar=False if your timeseries is 7 days a week, e.g. cryptocurrencies.

Parameters

**`use_stock_market_calendar` : bool** True for standard stock market calendar. False for trading 7 days a week.

Returns

**None**

### Function `stats`

```
def stats(
    ts,
    tlog,
    dbal,
    capital
)
```

Compute trading stats.

Parameters

**`ts` : pd.DataFrame** The timeseries of a symbol.
**`tlog` : pd.DataFrame** The trade log.
**`dbal` : pd.DataFrame** The daily balance.
**`capital` : int** The amount of money available for trading.

Examples

```
>>> stats = pf.stats(ts, tlog, dbal, capital)
```

Returns

**`stats` : pd.Series** The statistics for the strategy.

#### Function `summary`

```
def summary(
    stats,
    benchmark_stats=None,
    metrics=('annual_return_rate', 'max_closed_out_drawdown', 'best_month', 'worst_month', 'sha
    extras=None
)
```

Returns stats summary.

stats() must be called before calling this function.

Parameters

**`stats` : pd.Series** Statistics for the strategy.
**`benchmark_stats` : pd.Series, optimal** Statistics for the benchmark (default is None, which implies that a benchmark is not being used).
**`metrics` : tuple, optional** The metrics to be used in the summary (default is default_metrics).
**`extras` : tuple, optional** The extra metrics to be used in the summary (default is None, which imples that no extra metrics are being used).

## Module `pinkfish.stock_market_calendar`

Past and Future dates when the stock market is open from 1928 to 2024.

## Module `pinkfish.trade`

Trading agent.

### Classes

#### Class `DailyBal`

```
class DailyBal
```

Log for daily balance.

Initialize instance variables.

#### Attributes

**`_l` : list of tuples** The list of daily balance tuples.

#### Methods

#### Method `append`

```
def append(
    self,
    date,
    high,
    low,
    close
)
```

Append a new entry to the daily balance log.

Parameters

**`date` : str** The current date.
**`high` : float** The balance high value of the day.
**`low` : float** The balance low value of the day.

`close : float` The balance close value of the day.

Returns

**None**

### Method `get_log`

```
def get_log(
    self,
    tlog
)
```

Return the daily balance log.

The daily balance log consists of the following columns: 'date', 'high', 'low', 'close', 'shares', 'cash', 'leverage'

Parameters

`tlog : pd.DataFrame` The trade log.

Returns

`dbal : pd.DataFrame` The daily balance log.

### Class `Direction`

```
class Direction
```

The direction of the trade. Either LONG or SHORT.

#### Class variables

#### Variable `LONG`

#### Variable `SHORT`

### Class `Margin`

```
class Margin
```

The type of margin. CASH, STANDARD, or PATTERN_DAY_TRADER.

#### Class variables

#### Variable `CASH`

#### Variable `PATTERN_DAY_TRADER`

#### Variable `STANDARD`

### Class `TradeLog`

```
class TradeLog(
    symbol,
    reset=True
)
```

The trade log for each symbol.

Initialize instance variables.

**Parameters**

`symbol` : **str** The symbol for a security.
`reset` : **bool, optional** Use when starting new portfolio construction to clear the dict of TradeLog instances (default is True).

**Attributes**

`symbol` : **str** The symbol for a security.
`shares` : **int** Number of shares of the symbol.
`direction` : **pf.Direction** The direction of the trade, Long or Short.
`ave_entry_price` : **float** The average purchase price per share.
`cumul_total` : **float** The cumulative total profits (loss).
`_l` : **list of tuples** The list of matching entry/exit trade pairs. This list will become the official trade log.
`_raw` : **list of tuples** The list of raw trades, either entry or exit.
`open_trades` : **list** The list of open trades, i.e. not closed out.

**Class variables**

**Variable `buying_power`**

float : Buying power for Portfolio class.

**Variable `cash`**

int : Current cash, entire portfolio.

**Variable `instance`**

dict of pf.TradeLog : dict (key=symbol) of TradeLog instances used in Portfolio class.

**Variable `margin`**

float : Margin percent.

**Variable `multiplier`**

int : Applied to profit calculation. Used only with futures.

**Variable `seq_num`**

int : Sequential number used to order trades in Portfolio class.

**Instance variables**

**Variable `num_open_trades`**

Return the number of open orders, i.e. not closed out.

**Methods**

**Method `adjust_percent`**

```
def adjust_percent(
    self,
    date,
    price,
    weight,
    direction='LONG'
)
```

Adjust position to a target percent of the current portfolio value.

If the position doesn't already exist, this is equivalent to entering a new trade. If the position does exist, this is equivalent to entering or exiting a trade for the difference between the target percent and the current percent.

Parameters

**`date`** **: str** The trade date.
**`price`** **: float** The current price of the security.
**`shares`** **: int** The requested target weight.
**`direction`** **: pf.Direction, optional** The direction of the trade (default is Direction.LONG).

Returns

**int** The number of shares bought or sold.

### Method `adjust_shares`

```
def adjust_shares(
    self,
    date,
    price,
    shares,
    direction='LONG'
)
```

Adjust a position to a target number of shares.

If the position doesn't already exist, this is equivalent to entering a new trade. If the position does exist, this is equivalent to entering or exiting a trade for the difference between the target number of shares and the current number of shares.

Parameters

**`date`** **: str** The trade date.
**`price`** **: float** The current price of the security.
**`shares`** **: int** The requested number of target shares.
**`direction`** **: pf.Direction, optional** The direction of the trade (default is Direction.LONG).

Returns

**int** The number of shares bought or sold.

### Method `adjust_value`

```
def adjust_value(
    self,
    date,
    price,
    value,
    direction='LONG'
)
```

Adjust a position to a target value.

If the position doesn't already exist, this is equivalent to entering a new trade. If the position does exist, this is equivalent to entering or exiting a trade for the difference between the target value and the current value.

Parameters

**`date`** **: str** The trade date.
**`price`** **: float** The current price of the security.
**`shares`** **: int** The requested target value.
**`direction`** **: pf.Direction, optional** The direction of the trade (default is Direction.LONG).

Returns

**int** The number of shares bought or sold.

### Method `buy`

```
def buy(
    self,
    entry_date,
    entry_price,
    shares=None
)
```

Enter a trade on the long side.

Parameters

**`entry_date` : str** The entry date.
**`entry_price` : float** The entry price.
**`shares` : int, optional** The number of shares to buy (default is None, which implies buy the maximum number of shares possible with available buying power).

Returns

**int** The number of shares bought.

Notes

The 'buy' alias can be used to call this function for increasing or opening a long position.

### Method `buy2cover`

```
def buy2cover(
    self,
    exit_date,
    exit_price,
    shares=None
)
```

Exit a trade on the short side, i.e. buy to cover.

Parameters

**`exit_date` : str** The exit date.
**`exit_price` : float** The exit price.
**`shares` : int** The number of shares to buy to cover (default in None, which implies close out the short shares).

Returns

**int** The number of shares bought.

### Method `calc_buying_power`

```
def calc_buying_power(
    self,
    price
)
```

Calculate buying power.

### Method `calc_shares`

```
def calc_shares(
    self,
    price,
```

```
        cash=None
    )
```

Calculate shares using buying power before enter_trade().

Parameters

**price : float** The current price of the security.
**cash : float, optional** The requested amount of cash used to buy shares (default is None, which implies use all available cash).

Returns

**value : float** The number of shares that can be purchased with requested cash amount.

### Method `enter_trade`

```
    def enter_trade(
        self,
        entry_date,
        entry_price,
        shares=None
    )
```

Enter a trade on the long side.

Parameters

**entry_date : str** The entry date.
**entry_price : float** The entry price.
**shares : int, optional** The number of shares to buy (default is None, which implies buy the maximum number of shares possible with available buying power).

Returns

**int** The number of shares bought.

Notes

The 'buy' alias can be used to call this function for increasing or opening a long position.

### Method `equity`

```
    def equity(
        self,
        price
    )
```

Return the equity which is the total value minus loan. Loan is negative cash.

### Method `exit_trade`

```
    def exit_trade(
        self,
        exit_date,
        exit_price,
        shares=None
    )
```

Exit a trade on the long side.

Parameters

**exit_date : str** The exit date.
**exit_price : float** The exit price.
**shares : int, optional** The number of shares to sell (default is None, which implies sell all the shares).

Returns

**int** The number of shares sold.

Notes

The 'sell' alias can be used to call this function for reducing or closing out a long position.

### Method `get_log`

```
def get_log(
    self,
    merge_trades=False
)
```

Return the trade log.

The trade log consists of the following columns: 'entry_date', 'entry_price', 'exit_date', 'exit_price', 'pl_points', 'pl_cash', 'qty', 'cumul_total', 'direction', 'symbol'.

Parameters

**`merge_trade` : bool, optional** True to merge trades that occur on the same date (default is False).

Returns

**`tlog` : pd.DataFrame** The trade log.

### Method `get_log_raw`

```
def get_log_raw(
    self
)
```

Return the raw trade log.

The trade log consists of the following columns: 'date', 'seq_num', 'price', 'shares', 'entry_exit', 'direction', 'symbol'.

Returns

**`rlog` : pd.DataFrame** The raw trade log.

### Method `get_price`

```
def get_price(
    self,
    row,
    field='close'
)
```

Return price given row and field.

Parameters

**`row` : pd.Series** The timeseries of the portfolio.
**`field` : str, optional `{'close', 'open', 'high', 'low'}`** The price field (default is 'close').

Returns

**`price` : float** The current price.

### Method `get_prices`

```
def get_prices(
    self,
    row,
    fields=['open', 'high', 'low', 'close']
)
```

Return dict of prices for all symbols given row and fields.

Parameters

**`row` : pd.Series** The timeseries of the portfolio.
**`fields` : list, optional** The list of fields to use (default is [‘open’, ‘high’, ‘low’, ‘close’]).

Returns

**`d` : dict of floats** The price indexed by fields.

### Method `leverage`

```
def leverage(
    self,
    price
)
```

Return the leverage factor of the position given current price.

### Method `sell`

```
def sell(
    self,
    exit_date,
    exit_price,
    shares=None
)
```

Exit a trade on the long side.

Parameters

**`exit_date` : str** The exit date.
**`exit_price` : float** The exit price.
**`shares` : int, optional** The number of shares to sell (default is None, which implies sell all the shares).

Returns

**int** The number of shares sold.

Notes

The ‘sell’ alias can be used to call this function for reducing or closing out a long position.

### Method `sell_short`

```
def sell_short(
    self,
    entry_date,
    entry_price,
    shares=None
)
```

Enter a trade on the short side.

Parameters

**`entry_date` : str** The entry date.
**`entry_price` : float** The entry price.
**`shares` : int** The number of shares to sell short (default in None, which implies to sell short the maximum number of shares possible).

Returns

**int** The number of shares sold short.

#### Method `share_percent`

```
def share_percent(
    self,
    price
)
```

Return the share value as a percentage of total funds.

#### Method `share_value`

```
def share_value(
    self,
    price
)
```

Return the total value of shares of the security.

Parameters

**`price : float`** The current price of the security.

Returns

**`value : float`** The share value.

#### Method `total_funds`

```
def total_funds(
    self,
    price
)
```

Return the total account funds for trading given current price.

#### Method `total_value`

```
def total_value(
    self,
    price
)
```

Return the total value which is the total share value plus cash.

Parameters

**`price : float`** The current price of the security.

Returns

**`value : float`** The total value.

#### Class `TradeState`

```
class TradeState
```

The trade state of OPEN, HOLD, or CLOSE.

In the Daily Balance log, trade state is given by these characters: OPEN=‘O’, HOLD=‘-’, and CLOSE=‘X’

#### Class variables

#### Variable `CLOSE`

#### Variable `HOLD`

#### Variable `OPEN`

# Module `pinkfish.utility`

Utility functions.

## Variables

#### Variable `ROOT`

str: pinkfish project root dir.

## Functions

#### Function `find_nan_rows`

```
def find_nan_rows(
    ts
)
```

Return a dataframe with the rows that contain NaN values.

This function can help you track down problems with a timeseries. You may need to call `pd.set_option("display.max_columns", None)` at the top of your notebook to display all columns.

Examples

```
>>> pd.set_option("display.max_columns", None)
>>> df = pf.find_nan_rows(ts)
>>> df
```

#### Function `import_strategy`

```
def import_strategy(
    strategy_name,
    top_level_dir='examples',
    module_name='strategy'
)
```

Import a strategy from a python .py file.

Parameters

**`strategy_name` : str** The leaf dir name that contains the strategy to import.
**`top_level_dir` : str, optional** The top level dir name for the strategies (default is 'examples').
**`module_name` : str, optional** The name of the python module (default is 'strategy').

Returns

**module** The imported module.

Examples

```
>>> strategy = import_strategy(strategy_name='190.momentum-dmsr-portfolio')
```

#### Function `is_last_row`

```
def is_last_row(
    ts,
    index
)
```

Return True for last row, False otherwise.

### Function `print_full`

```
def print_full(
    x
)
```

Print every row of list-like object.

### Function `read_config`

```
def read_config()
```

Read pinkfish configuration.

### Function `set_dict_values`

```
def set_dict_values(
    d,
    value
)
```

Return dict with same keys as d and all values equal to 'value'.

### Function `sort_dict`

```
def sort_dict(
    d,
    reverse=False
)
```

Return sorted dict; optionally reverse sort.

---

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).