Testing & Trading Strategies

on Forex Markets

using FxEngine

Version history:

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Update** |
| 0.1 | 2016-07-23 | Initial version |
|  |  |  |
|  |  |  |
|  |  |  |

Index

[1. Purpose 3](#_Toc457059427)

[2. The trading process 3](#_Toc457059428)

[2.1 basics 3](#_Toc457059429)

[3. How FxEngine works 4](#_Toc457059430)

[3.1 Getting started 5](#_Toc457059431)

[3.2 Imported modules 6](#_Toc457059432)

[3.3 Threads (import threading) 7](#_Toc457059433)

[3.4 Logging (import logging) 7](#_Toc457059434)

[3.5 Program structure 9](#_Toc457059435)

[Glossary 12](#_Toc457059436)

[References 13](#_Toc457059437)

# Purpose

With FxEngine you can:

1. **test** your strategy on historical data (“backtesting”)
2. **trade** your strategy in the market (“trading”)

# The trading process

## 2.1 basics

In order to understand how FxEngine works, we have to understand the **trading process** first.

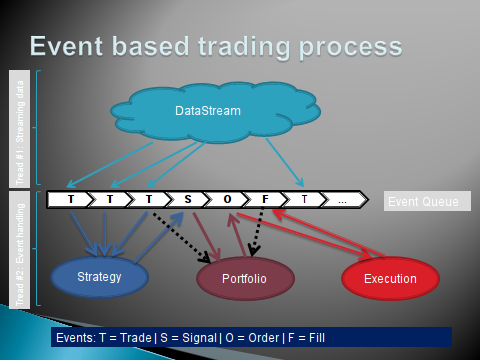
First prices are generated (so called TICK events). Next, several steps are performed that may (or may not) result in an order to be executed.

*Generating prices*Based on supply and demand in the market, a TICK events containing the last price information, are generated (and put on the queue).

*Event handling*An order is not executed until it passes several steps successfully performed by the following entities Strategy (manager), Portfolio (manager) and Execution (manager). Depending on the type of Event in the Queue these entities perform the following actions:

1. **TICK event?** 
   1. the Strategy Manager can decide to trigger a SIGNAL event (and puts in on the queue)
   2. the Portfolio Manager updates the value of his portfolio
2. **SIGNAL event?**
   1. The Portfolio Manager checks the SIGNAL, does in fit in risk management, if so the Portfolio manager triggers an ORDER event (and puts in on the queue)
3. **ORDER event?**
   1. The Execution Manager tries to execute the order at the given conditions. If the order can be executed successfully in the market, he triggers a FILL event (and puts it on the queue)
4. **FILL event?**
   1. The Portfolio Manager updates his position

**Figure 1: the trading process**



# How FxEngine works

FxEngine is designed to simulated the trading process described in chapter 1.

FxEngine is a 4-component event based backtester & trader running two threads:

* Thread #1: streams prices from a file or from a broker
* Thread #2: performs the event handling in the queue

The TICK events allow the two threads to communicate with each other.

The 4 components are:

1. DataStream (either historical data or market)
2. Strategy
3. Portfolio
4. Execution (either simulated or real)

The nice thing about this design is that the Strategy and the Portfolio components are reused for testing and trading. During testing, it tries to simulate the “real” trading process as realistic as possible.

## 3.1 Getting started

In order to run a program, you need:

* Appropriate environment variables:
  + PYTHONPATH
  + OANDA\_API\_ACCESS\_TOKEN (live trading only)
  + OANDA\_API\_ACCOUNT\_ID (live trading only)
* Appropriate settings.py:
  + DOMAIN (“real”, “practice” or “sandbox”)
  + BACKTEST (True or False)
  + BACKTESTFILE (e.g. test.csv)
  + INSTUMENTS (e.g. [“EUR\_USD”, “EUR\_ZAR”])
  + UNITS (e.g. 10000)
* Appropriate logging.conf:
  + [loggers]
  + [handlers]
  + [formatters]
* Appropriatre environment & directory

*Setting environment variables*

Open a terminal and type:

|  |
| --- |
| $ export PYTHONPATH=/home/pieter/projects #1  $ printenv PYTHONPATH #2 |

In the 1st line you created a shell variable PYTHONPATH and turned this shell variable into a environment variable by exporting PYTHONPATH. In the second line you check that PYTHONPATH has been set properly.

Next, you run:

|  |
| --- |
| $ workon qsforex $ cd ~/projects/fxengine $ python trading/trading.py |

## 3.2 Imported modules

The FxEngine packages imports several Python modules new to me. All modules are standard included in Python, except the request modules required for streaming data from the OANDA broker.

A description for each modules has been summarized in the tables below.

**Table 1: imports per component**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Imported module** | **FxEngine Component** | | | | |
| **streaming** | **strategy** | **portfolio** | **execution** | **trading** |
| **json** | **X** |  |  | **X** |  |
| **csv** | **X** |  |  |  |  |
| **time** | **X** |  |  |  | **X** |
| **logging** | **X** |  | **X** | **X** | **X** |
| **abc** | **X** |  | **X** | **X** |  |
| **requests** | **X** |  |  |  |  |
| **Queue** |  |  |  | **X** | **X** |
| **threading** |  |  |  |  | **X** |
| **random** |  | **X** |  |  |  |
| **copy** |  |  | **X** |  |  |
| **httplib** |  |  |  | **X** |  |
| **urllib** |  |  |  | **X** |  |

**Table 2: imports used in FxEngine package**

|  |  |
| --- | --- |
| **Module** | **Contains classes for…** |
| **json** | … parsing JSON (JavaScript Object Notation, see <http://json.org>) from strings or files into a Python dictionary or list and converting Python dictionaries or lists into JSON strings. |
| **csv** | … reading and writing Comma Separated Value (CSV) files |
| **time** | … manipulating time values |
| **logging** | …logging at various levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) |
| **abc** | …defining Abstract Base Classes (ABCs) |
| **Queue** | …LIFO en FIFO queues |
| **threading** | …emulating a subset of Java’s threading model |
| **random** | …generating random numbers, samples and permutations from several distributions. |
| **copy** | …performing (shallow and deep) copy operations |
| **httplib** | …implementing the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module urllib uses it to handle URLs that use HTTP and HTTPS.  Note: The requests package is recommended for a higher-level http client interface. |
| **urllib** | … providing a high-level interface for fetching data across the World Wide Web.  Note: The requests package is recommended for a higher-level http client interface. |
| **requests** | ….providing HTTP for humans. |

## 3.3 Threads (import threading)

Simply put, a thread is **a program's path of execution [1]**. Many programs run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this.

Fxengine runs two threads:

* Thread #1: streams prices from a file or from a broker in streaming/streaming.py
* Thread #2: performs Event handling to/from the queue

The two threads communicate with each other by means of the TickEvents generated by the datastream and put on the queue by the first thread.

## 3.4 Logging (import logging)

Fxengine logs events. This is cleaner than printing as logging does not “pollute” the code with print statement all over the place [2]. Furthermore, you can simply control logging output by modifying the INI-formatted “logging.conf” file.

There are 5 levels of logging (in ascending order):

* DEBUG
* INFO
* WARNING
* ERROR
* CRITICAL

If you set logging to the level DEBUG all messages are logged while setting logging to the level ERROR will only log ERROR and CRITICAL messages. Is summary, logging is performed according to the “error level hierarchy” stated above.

Controlling logging output:

|  |  |  |
| --- | --- | --- |
| Item | options | How |
| Output to… | file Console/stdout | logging.basicConfig(**filename="sample.log"**, level=logging.INFO)  logging.basicConfig(**filename=None**, level=logging.INFO) |
| Output level | DEBUG INFO WARNING ERROR  CRITICAL | logging.basicConfig(filename="sample.log", level=logging.**DEBUG**)  logging.basicConfig(filename="sample.log", level=logging.**INFO**)  etc. |
| Append or overwrite |  | logging.basicConfig(… filemode=’a’) or ‘w’ |
| Log exceptions |  | log = logging.getLogger("ex") |

* output: to file or console/stdout
* append or overwrite

By default, all logging messages are appended to the logging file. If you would rather have your logger overwrite the log, then pass in a filemode=’w’.

|  |
| --- |
| The Python logging module consists of four components: loggers, handlers, filters, and formatters.  Each of these items is controlled by the settings in the file **logging.conf**    As of [4] the four components perform the following tasks:   * Loggers expose the interface that application code directly uses. * Handlers send the log records (created by loggers) to the appropriate destination. * Filters provide a finer grained facility for determining which log records to output. * Formatters specify the layout of log records in the final output.   Next to a consoleHandler, I also included a fileHandler to have the logging ouput written to file (in stead of to screen). |

## 3.5 Program structure

**Folder: root**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| settings.py | --- | ---- | --- |
| logging.conf | --- | --- | --- |

**Folder: trading**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| trading.py | --- | --- | trade() |

**Folder: streaming**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| streaming.py | AbstractPriceStream | event\_queue stoprequest cur\_prices | \_\_init\_\_() stream\_to\_queue() |
| StreamingForexPrices\_OANDA(APS) | domain access\_token,  account\_id instruments, events\_queue stoprequest | \_\_init\_\_() connect\_to\_stream() stream\_to\_queue() |
| StreamingPricesFromFile(APS) | csv\_file  events\_queue cur\_prices stoprequest logger | \_\_init\_\_() stream\_to\_queue() |
| MockPriceStream(APS) | event\_queue stoprequest cur\_prices | \_\_init\_\_() newprice() stream\_to\_queue() |

**Folder: strategy**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| strategy.py | TestRandomStrategy | events ticks | \_\_init\_\_() calculate\_signals() |

**Folder: portfolio**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| portfolio.py | AbstractPortfolio | --- | execute\_tick\_event() execute\_signal\_event() execute\_fill\_event() |
| Portfolio(APS) | ticker events base="EUR", leverage=20 equity=100000.0 risk\_per\_trade=0.02  balance risk\_per\_trade trade\_units positions logger | \_\_init\_\_() calc\_risk\_position\_size() add\_new\_position() add\_position\_units()  remove\_position\_units() close\_position() execute\_close\_all\_positions()  execute\_tick\_event() execute\_signal\_event() execute\_fill\_event() |
| position.py | Position | side market units,  exposure avg\_price cur\_price | \_\_init\_\_() calculate\_pips() calculate\_profit\_base()  calculate\_profit\_perc() update\_position\_price() |

**Folder: execution**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| execution.py | AbstractExecution | --- | execute\_order() |
| MockExecution(AE) | event\_queue ticker | \_\_init\_\_() execute\_order() |
| ExecutionAtOANDA(AE) | domain access\_token account\_id event\_queue | \_\_init\_\_() obtain\_connection() execute\_order() |
| marketstate.py | MarketState(AE) | bid ask | \_\_init\_\_() update\_bid\_ask() |

**Folder: event**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module (c.q. file)** | **Class** | **Attributes** | **Methods** |
| event.py | Event | --- | --- |
| TickEvent(Event) | Instrument time bid ask | \_\_init\_\_() \_\_str\_\_() \_\_repr\_\_() |
| SignalEvent(Event) | Instrument order\_type side | \_\_init\_\_() \_\_str\_\_() \_\_repr\_\_() |
| OrderEvent(Event) | Instrument units order\_type side | \_\_init\_\_() \_\_str\_\_()  \_\_repr\_\_() |
| FillEvent(Event) | Instrument units side price | \_\_init\_\_() \_\_str\_\_()  \_\_repr\_\_() |

# Glossary

|  |  |  |
| --- | --- | --- |
| **item** | **Definition** | **Source** |
| **Environmental variables** | variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell | [3] |
| **Shell variables** | variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory | [3] |
|  |  |  |
|  |  |  |

# References

[1] <http://www.javaworld.com/article/2077138/java-concurrency/introduction-to-java-threads.html>

[2] <http://www.blog.pythonlibrary.org/2012/08/02/python-101-an-intro-to-logging/>

[3] <https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps>

[4] https://docs.python.org/3/howto/logging.html