Clojure for Finance

Starting with the importance of representing data and effectively performing calculations, this book takes you all the way to being competent in financial analytics and building financial applications.

We first introduce the notions of computation and finance. This will help you understand Clojure's ability to solve real-world problems in many domains, especially finance. Next, we show you how to implement a simple moving average algorithm, using some of Clojure's core data transformation functions. These functions are used to calculate and manipulate data.

You will then learn how to implement slightly more complicated equations, traverse data, and deal with branching and conditional dispatch. These and other concepts are built on top of each other, all guided by a strategy of using data as the interface between systems. Finally, you will discover how to build more advanced algorithms while manipulating and composing functions.

Who this book is written for

This book is for you if you're a finance professional, currently using VBA (Excel) to perform financial calculations and analysis. It shows you ways Clojure can help augment and improve your tools and processes. Basic knowledge of financial concepts is essential. Basic programming knowledge would be an added advantage.

What you will learn from this book

- Quickly and effectively represent data and perform calculations using Clojure
- Use Clojure's basic language tools, such as laziness, immutability, and first-class functions, to solve real-world problems
- Get acquainted with the advanced approaches to calculating and transforming data and building algorithms
- Use Clojure's functions to access, update, and compose data structures
- Be introduced to the concepts of macros, Domain-specific Languages, higher-order functions
- Compose simple and exponential functions together to get a buy or sell signal

Leverage the power and flexibility of the Clojure language for finance
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Developing the Simple Moving Average'
- A synopsis of the book’s content
- More information on Clojure for Finance
About the Author

Timothy Washington is a senior software developer with over 15 years of experience in designing and building enterprise web applications from end to end. His experience includes delivering stable, robust software architectures to organizations ranging from start-ups to Fortune 500 companies. His skills include managing agile projects, systems analysis and design, functional programming, DSL and language design, and object-oriented design, with contributions to the open source community.
Preface

Clojure is a dynamic programming language with an emphasis on functional programming. As a functional language with immutability and laziness as the default, Clojure is well suited to financial modeling. Such languages help developers work at high-levels of abstraction, thus implementing features much more quickly and with more confidence, than is otherwise possible in languages without these facilities.

In this book, we will demonstrate Clojure's utility in processing and analyzing financial data. Whether it's core banking, risk assessment, compliance, or other domains, this book's audience mostly comprises finance professionals who can use Clojure to improve their Excel spreadsheets or existing toolsets.

What this book covers

Chapter 1, Orientation – Addressing the Questions Clojure Answers, gives you the background required to understand Clojure's utility to solve real-world problems in many domains, and finance in particular. It also orients and helps you understand Clojure's approach to computation.

Chapter 2, First Principles and a Useful Way to Think, provides you with an understanding of what a stream of stock price data may look like. It then applies some core functional programming features that Clojure provides to transform data into the stock price shape we desire.

Chapter 3, Developing the Simple Moving Average, aims to teach you how to translate requirements to data input, deriving the target data output, and reasoning about an equation needed to achieve your output. It also helps you understand Vars and bindings and how to implement a Simple Moving Average (SMA). This chapter focuses on developing a solid understanding of a problem, easily representing data, and quickly performing calculations.
Chapter 4, Strategies for Calculating and Manipulating Data, implements two more functions, the Exponential Moving Average (EMA) and Bollinger Band. These are both technical trading indicators, which involve more advanced mathematics. This chapter explores the math in each equation and the algorithmic steps needed to arrive at a desired endpoint.

Chapter 5, Traversing Data, Branching, and Conditional Dispatch, provides you with an understanding of more advanced branching and conditional logic. This chapter fleshes out the overall approach of Clojure's functions around recursion, list comprehensions, conditional dispatch, and first-order functions. This should then enable you to traverse and manipulate your data until you get what you want.

Chapter 6, Surveying the Landscape, takes a step back and reviews all of Clojure's features together. We'll review Clojure's scalars, collection types, and composite data types. Then, we'll take a look at how to use Clojure's functions to access, update, and compose data structures, I/O operations, and ways to approach what are known as side effects. We will also touch on Clojure's options when dealing with concurrency and parallelism.

Chapter 7, Dealing with Side Effects, shows how a componentized architecture can encapsulate all the functionality that we've developed so far. We will also develop a persistence strategy to write out our core tick list and accompanying analytics in the EDN format.

Chapter 8, Strategies for Using Macros, helps you understand how to read in data using Clojure. Once data is in our system, we will look at a few ways of querying it, including simple filtering and adding more constraints in a logical OR and AND fashion. We will also derive a little query language using macros.

Chapter 9, Building Algorithms – Strategies to Manipulate and Compose Functions, uses all the knowledge you've gained so far to design buy and sell signals. It teaches you how to structure your data for further analysis, refactor your lagging indicator functions (SMA, EMA, and Bollinger Bands) to work lazily, and compose them together to get new information.
Developing the Simple Moving Average

We can think of humans as a symbolic species. By this, I mean that we use a number of characters and scripts to represent abstract things. These symbols are used by us to communicate with each other, our outside world, and our abstract thoughts. Symbolic representation is used in written language, math, and music notation. Dolphins or monkeys, while very intelligent, do not by themselves use any written symbols to communicate with each other. I’m not mentioning this simply as a philosophical musing. I think this is directly useful to consider when reaching for ideas on how to fully capture and represent our data.

In this chapter, we’ll cover the following topics:

- Translating requirements to data input
- Knowing the target data output
- Reasoning an equation needed to achieve our output
- Understanding Vars and bindings
- Working with lazy sequences
- Implementing a Simple Moving Average (SMA) equation
- Destructuring
Perception and representation

A big part of building working systems is the ability to correctly and fully abstract the problem we're trying to solve. In our case, this abstraction means quantifying all the input data that a problem involves, knowing the precise result data we need, and any processes or transformations that affect these inputs to get your desired output. This is what I'll describe as fully perceiving a problem. Let's take Clojure data structures, functions, and FP approaches as a way of representing our problem and solving it.

We ultimately need to gauge how close our perception and representations are to actual stock price data. However, for now, we have an infinite stream of price and time points. Apart from this stream of data, we want to calculate a moving average of prices. So, an average is just the sum of a collection of things divided by the length of the collection. This sounds easy enough. However, the moving qualifier only means that the average is calculated at each tick increment of price/time data.

Knowing the data input

When we completed Chapter 2, First Principles and a Useful Way to Think, we had helper functions that generated some test time series data for us. So, our input data will look something like the following output. This is just a lazy sequence of maps. Each map has two entries with the keys, last-trade-price and last-trade-time. The fact that the input data is a lazy sequence is important. It means we can treat it as infinite, so it can be consumed as the data comes in:

```clojure
[{:last-trade-time #inst "2015-09-24T04:13:13.868-00:00",
  :last-trade-price {:last 5.46160487301605}},
{:last-trade-time #inst "2015-09-24T04:13:15.868-00:00",
  :last-trade-price {:last 6.540895364039775}}
{:last-trade-time #inst "2015-09-24T04:13:16.868-00:00",
  :last-trade-price {:last 5.53301182972796}}
{:last-trade-time #inst "2015-09-24T04:13:17.868-00:00",
  :last-trade-price {:last 5.827927905654936}}
{:last-trade-time #inst "2015-09-24T04:13:19.868-00:00",
  :last-trade-price {:last 6.31043832017862}}
{:last-trade-time #inst "2015-09-24T04:13:21.868-00:00",
  :last-trade-price {:last 7.1373823393671865}}
{:last-trade-time #inst "2015-09-24T04:13:24.868-00:00",
  :last-trade-price {:last 8.564858807240624}}
{:last-trade-time #inst "2015-09-24T04:13:24.868-00:00",
  :last-trade-price {:last 10.27783056888747}}
{:last-trade-time #inst "2015-09-24T04:13:25.868-00:00",
  :last-trade-price {:last 8.222264454950999}}
]
Knowing the data output

The output that we ultimately want is a running average of the last few ticks of our time series. For the number of our last few ticks, I'm going to pick 20 because it's a nice round number. However, we'll make our algorithm flexible enough to accept different time intervals for our running average. So, now it looks like our output can start with the input with an added entry for the running average at each tick, which is described in the following output. We can't evaluate this data structure yet. The elided values after the :average key, are just pseudo code, representing the result we need to reach:

```clojure
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:13.868-00:00",
 :last-trade-price {:last 9.86617345941199}}
...
```

Here's a sample graph output (your data may vary):
Developing the Simple Moving Average

```Clojure
{:last-trade-time #inst "2015-09-24T04:13:15.868-00:00",
 :last-trade-price {:last 6.540895364039775}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:16.868-00:00",
 :last-trade-price {:last 5.53301182972796}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:17.868-00:00",
 :last-trade-price {:last 5.827927905654936}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:19.868-00:00",
 :last-trade-price {:last 6.31043832017862}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:21.868-00:00",
 :last-trade-price {:last 7.1373823393671865}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:24.868-00:00",
 :last-trade-price {:last 8.564858807240624}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:24.868-00:00",
 :last-trade-price {:last 10.277830568688747}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:25.868-00:00",
 :last-trade-price {:last 8.222264454950999}}
{:average <...>,
 :last-trade-time #inst "2015-09-24T04:13:28.868-00:00",
 :last-trade-price {:last 9.866717345941199}}
...}
```

Reasoning about the equation needed to achieve our output

Our simple moving average equation will operate for an infinite input stream of ticks. We'll pick a starting point anywhere in the infinite input stream, pick a size \( n \) that we'd like to average, take the previous \( n \) (including the current) ticks, and apply our average equation.

Nominally, our equation will be similar to \(/ \text{price-sum} (\text{count ticks-within-window})\).

Here, \text{price-sum}, as the name implies, is the sum of all the prices with our chosen average range, and \text{ticks-within-window} is the collection of ticks we want to include in our calculations.

The important part is that this equation has to be applied at each increment of the tick window. To get an average, therefore, we have to use the length of the tick window as the starting point.
So, we'll begin our moving average function by passing in the infinite lazy sequence (tick-seq) and the size of ticks for which we want to calculate our average (tick-window). The next thing to note in the following code, is that I've introduced a new core Clojure function, partition, which can be found at http://clojuredocs.org/clojure.core/partition. The partition function conveniently divides many kinds of collections.

So, in your REPL, try evaluating `partition 4 (range 20)`. You should see the following result. We can see that it's dividing a list of 20 into five lists with four elements each, for example, [0-indexed (0 to 19)]:

```
(partition 4 (range 20))
;;=> ((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15) (16 17 18 19))
```

Our case, though, requires that each partition's starting index begins at the next increment from the previous one. The partition function also provides this feature. If you try evaluating `partition 5 1 (range 10)`, you should see the following result:

```
(partition 5 1 (range 10))
;;=> ((0 1 2 3 4) (1 2 3 4 5) (2 3 4 5 6) (3 4 5 6 7) (4 5 6 7 8) (5 6 7 8 9))
```

With this knowledge, we can start by dividing the infinite sequence that we've been given. Try creating this function now:

```
(defn moving-average 
[tick-seq tick-window]
(partition tick-window 1 tick-seq))
```

Then, evaluating the following expressions should yield results like in the subsequent example code. You'll notice that the second price in the first list, (15.423964294614002, in my case), is the first price in the second list. This is the partition window sliding along one increment:

```
(def pricelist (generate-prices 5 15))
(def timeseries (generate-timeseries pricelist))
(def our-average (moving-average timeseries 20))
(take 2 our-average)
;;=> (((:last-trade-price {:last 5.466160487301605},
  :last-trade-time #inst "2015-09-24T04:13:13.868-00:00")
  {:last-trade-price {:last 6.540895364039775},
   :last-trade-time #inst "2015-09-24T04:13:15.868-00:00"})
  {:last-trade-price {:last 5.53301182972796},
   :last-trade-time #inst "2015-09-24T04:13:16.868-00:00"}
  ...
```

---

33
Developing the Simple Moving Average

(  {:last-trade-price {:last 6.540895364039775},
    :last-trade-time #inst "2015-09-24T04:13:15.868-00:00"}
  {:last-trade-price {:last 5.53301182972796},
    :last-trade-time #inst "2015-09-24T04:13:16.868-00:00"}
  ...
)

Understanding Vars and bindings
The previous expressions work. However, there's some assumed knowledge that may be helpful to deconstruct. Firstly, I'm using def to bind a value (lazy sequences, in this case) to a symbol. To be precise, these values are called Vars. Vars are just a way to bind a named location (a symbol) to something else, such as numbers, functions, or in our case, lazy sequences. The notion of binding is important.

It allows us to use a value on a per context basis, that is, we have the ability to dynamically rebind Vars in different contexts. So, the following code is possible (taken from http://clojure.org/vars):

```clojure
(def ^:dynamic x 1)
(def ^:dynamic y 1)
(+ x y)
;;=> 2
(binding [x 2 y 3]
  (+ x y))
;;=> 5
(+ x y)
;;=> 2
```

The root context is the namespace in which we evaluate our expressions. By default, the user namespace along with Clojure's core functions (which are bound to the Vars symbol), are always available. The def, defn, and with-redefs functions allow us to rebind Vars in a local or root context (of our namespace). Vars and context binding are important because they give us the ability to bind different functions (or entire components or programs) to a symbol for a specific context. This includes redefining functions in a running program. So, for example, you could wrap a function with logging behavior only in certain call contexts (or threads that aren't discussed here). For further information on Vars, visit http://clojure.org/vars.
Working with lazy sequences

There's another subtle notion to grasp when we talk about infinite lazy sequences. If we normally assign (or bind, in this case) some list or collection, we expect those artifacts to have values right away. You may have noticed that even after binding pricelist and timeseries, nothing happened. This is to say that binding these two Vars didn't immediately compute any values from the aforementioned expressions. This is the main feature that laziness provides. Only when we needed to pull out values using, for example, \((\text{take } 40 \ (\text{generate-prices } 5 \ 15))\) or \((\text{take } 2 \ \text{our-average})\), were the expressions actually run. This means that \((\text{def timeseries} \ (\text{generate-timeseries} \ \text{pricelist}))\) bound a lazy sequence to the timeseries symbol. We can also pass round this lazy sequence without necessarily consuming it.

In *Down The Clojure Rabbit Hole*, Clojure developer, Christophe Grand, describes sequences as "Reified Computation" and as a core part of Clojure's computation model, which can be seen at [http://www.infoq.com/presentations/clojure-stories](http://www.infoq.com/presentations/clojure-stories). By reified, what he's referring to is to literally realize computation. This allows us to treat sequences as a data flow pipeline. This just means that we can compose functions, and pieces of functions with greater ease as long as everybody's using lazy sequences.

The idea here is that when data enters your system from the outside world, it is mostly a more concrete form (such as lists, vectors, and so on). However, most core Clojure functions optimally consume and produce sequences. You may have also noticed that most of the core functions we've used so far (repeatedly, filter, map, and so on) also produce sequences. If we can begin to visualize data flowing through our systems, it becomes very useful to swap different functions that can perform anything from analysis to transformation and so on. It allows what software developers describe as *loose coupling* ([http://en.wikipedia.org/wiki/Loose_coupling](http://en.wikipedia.org/wiki/Loose_coupling)) between functions and components. We can thus allow data and its desired outputs to drive design.

Implementing our equation

So far, our *moving-average* function only gives us a partitioned list of prices (or ticks). Now, we only need to visit a sliced list at each increment and calculate each increment's price average. The reduce function provides a way of accumulating (or folding over) each item in a list and producing a result based on those values. We'll take our partitioned list and have reduce apply an accumulator function on each sublist. The form looks roughly like this:

\[
(\text{reduce} \ (\text{fn} \ [\text{accumulated-result-so-far} \ \text{each-item-list}] \\
\quad \quad ;; \ \text{** single line comments come after semi-colons}
\)
Developing the Simple Moving Average

;; do stuff, return accumulated result
;; for each item iteration
)

initial-list-of-accumulated-values
initial-list-of-input-values)

Our accumulator function only has to maintain a few temporary values that our equation needs. In this case, it's the sum of each-item-list (or lazy sequence). For this, we want to contain or bind intermediate values in a temporary context. The let macro gives us a scope or temporary place to set and use some vars.

To begin with, let's get a visualization of how our list will be traversed. We want to look at 20 price points or ticks at a time. Since this is a running average, our function will need to take a look at the first 20 ticks before getting the first average. By doubling the amount of ticks given to partition, it will give you a list of the first 20. It will then increment this number by 1 until the 40th tick (39) is reached. Also, since partition doesn't want to give you a list that contains less than 20 ticks, it will discard any list above this number. Thus, by constraint, some lists only have 19 items at the most, that is, a partitioned list at the 20th point will have the remaining ticks up to 39:

(partition 20 1 ' (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39))
'((0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21)
(3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
(4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24)
(6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25)
(7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26)
(8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27)
(9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28)
(10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32)
(14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33)
(15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35)
(17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36)
(18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37)
(19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38)
(20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39))
This is how partition will behave in the following function. We need to double the given tick window and remember to take this amount as the input in an infinite sequence. When reducing over each partitioned list, we calculate the sum, and then the average. With the average for our given list in place, we'll add it to the existing last-trade-price and last-trade-time (merge and zipmap). We'll then add these to the tail position of our list (using the function lazy-cat). Let's write our simple-moving-average function now. In your leiningen project, create a new file called analytics.clj under the directory structure, src/edgar. The final file path should be src/edgar/analytics.clj. At the top of this file, declare the namespace with the (ns edgar.analytics) form. In here, we'll also want to use the code from our edgar.core namespace. Here's how we can do this:

```clojure
(ns edgar.analytics
  (:require [edgar.core :as core]))
```

Now we can start to write our function as follows:

```clojure
(defn simple-moving-average
  "This is an optional documentation string that can be passed to your functions definition. Here, you can put any relevant information about the function and its definition. For example, we can note down the meaning of the 'option' argument."
  Options are:
  :input - input key function will look for (defaults to :last-trade-price)
  :output - output key function will emit (defaults to :last-trade-price-average)
  :etal - other keys to emit in each result map
  ** This function assumes the latest tick is on the left**
  [options tick-window tick-list]
  (let [start-index tick-window
        {input-key :input
         output-key :output
         etal-keys :etal
         :or {input-key :last-trade-price
              output-key :last-trade-price-average
              etal-keys [:last-trade-price :last-trade-time]}}
        options]
    (reduce (fn [rslt ech]
```
Developing the Simple Moving Average

(let [tsum (reduce (fn [rr ee]
                        (let [ltprice (:last-trade-price ee)]
                          (+ ltprice rr))) 0 ech)
         taverage (/ tsum (count ech)))

(lazy-cat rslt
    [(merge
        (zipmap etal-keys
            (map #(% (last ech)) etal-keys))
        {output-key taverage :population ech}))])

Let's inspect this slowly for good measure:

- All the code takes place inside a `let` binding (http://clojuredocs.org/clojure.core/let). In Clojure, `let` is a special function that lets us assign values to symbols that we can use inside the `let` block. So, `start-index` is assigned the value of `tick-window`.

- The next expression is a trick called destructuring (https://gist.github.com/john2x/e1dca953548bfdfb9844). This is simply a technique to extract values from a collection or map and assign them to symbols. So, in our preceding example, the values from the `options` map are extracted with the `:input`, `:output`, and `:etal` keys. These values are assigned to the symbols, `input-key`, `output-key`, and `etal-keys`, respectively. If any of the keys, such as `:input`, `:output`, or `:etal`, cannot be found, then default values of `:last-trade-price`, `:last-trade-price-average`, or `[:last-trade-price :last-trade-time]` are assigned.

- Our inner `map` (within `lazy-cat`) uses an anonymous `#()` function, where the `%` sign is an implicit parameter or each item is mapped over.

- Recall that our input data looks like the following list. Here, `ech`, an argument of the reducing function, represents each list that the `partition` function generated:

```
{[:last-trade-price {:last 11.08212939452855},
  :last-trade-time #<DateTime 2015-04-19T17:56:57.596Z>}
{[:last-trade-price {:last 15.423964294614002},
  :last-trade-time #<DateTime 2015-04-19T17:56:58.596Z>}
{[:last-trade-price {:last 15.737626638301053},
  :last-trade-time #<DateTime 2015-04-19T17:57:00.596Z>}
```

[38]
etal-keys is a [:last-trade-price :last-trade-time] vector that is created in the let block's binding form. So, the (map #(% (last ech)) etal-keys) expression maps an anonymous function over this vector. Recall that keywords also act as functions on maps. This means that we're calling :last-trade-price and :last-trade-time on a map, that looks like 
{:last-trade-price {:last 11.08212939452855}, :last-trade-time #inst "2015-09-24T04:13:13.868-00:00" }.

The (map #(% (first ech)) etal-keys) expression pulls out a list of values that look like 
({:last 5.466160487301605} #inst "2015-09-24T04:13:13.868-00:00")
This is done for each element that is reduced over.

We'll then rejoin the aforementioned values to the extraction keys that we just used. The zipmap function does this by interleaving values in two or more lists to produce a map:

(def etal-keys [:last-trade-price :last-trade-time])

(map #(% {:last-trade-price {:last 5.466160487301605}, :last-trade-time #inst "2015-09-24T04:13:13.868-00:00"}))
etal-keys
;; {:last-trade-time #inst "2015-09-24T04:13:13.868-00:00", :last-trade-price {:last 5.466160487301605}}
There is an actual `interleave` function that produces a result list (though this is not what we want). We want to produce a result map, which is what the `zipmap` function provides:

```lisp
(interleave '(1 2 3 4) '(:a :b :c :d))
;; (1 :a 2 :b 3 :c 4 :d)
```

We'll then take the map result and merge in our new average value. The `merge` function is simple enough, simply joining together the values of two maps. We can then employ this function to merge together the result of the `zipmap` call to the new `zipmap` entry that we want to add:

```lisp
(merge {:a 1 :b 2} {:c 3})
;; {:c 3, :b 2, :a 1}
```

```lisp
(merge
  (zipmap etal-keys
    (map #(% {:last-trade-price 6.111604269033163, :last-trade-time trade-time})
      etal-keys))
  {:last-trade-price-average 6.0972436})
;; {:last-trade-price-average 6.0972436,
   ;; :last-trade-time #inst "2015-06-28T18:31:51.810-00:00",
   ;; :last-trade-price 6.111604269033163}
```

Adding the result of this to our running (reduced) list is a bit trickier. There's a small semantic inefficiency that we have to manage. Our running time series is infinite and the newest values on the right. The `reduce` function is also known as a left fold (see https://en.wikipedia.org/wiki/Fold_(higher-order_function)), which means that it collects values from left to right. So far, we've been able to blithely ignore the ordering of our lists. However, if we `cons` or `conj` an item into the result list, it will always go to the head of the list, which is on the left. Recall that the list implementation decides how to append items based on its own knowledge of what's efficient. In this case, the left placement happens to be the most efficient way to append an item. If each `rslt` collection were a vector, then `conj` would add the new item to the end (or right) of the vector, instead of adding it to the head (or left) of it. This is just an alternative path to keep in mind:

```lisp
(conj '(1 2 3) 4)
;; (4 1 2 3)

(cons 4 '(1 2 3))
;; (4 1 2 3)
```
Since we are dealing with a time series, however, the ordering of the list is crucially important. Thus, we need to place the result to the right-hand side of the list. This is tricky because we'll need to switch the kind of collection in which we place the result so that it goes to the collection's end. The \texttt{concat} function joins the contents of two lists and puts the addition on the right-hand side. The \texttt{lazy-cat} macro does the same thing but respects the semantics of lazy sequences, only realizing values when invoked. The key thing here is that both parameters need to be lists (or vectors). If we pass only a map to \texttt{lazy-cat}, it will do the next best thing it knows, which is to get the lists to join together with the sequenced out the entries in the map:

\begin{verbatim}
(concat [1 2] [3]) ;; (1 2 3)

(lazy-cat [1 2] [3]) ;; (1 2 3)

;; Not what we want
(lazy-cat '(1)
   (merge
    (zipmap etal-keys
       (map #(% {:last-trade-price 6.111604269033163, :last-trade-time trade-time})
        etal-keys))
    {:last-trade-price-exponential 6.0972436}))

(seq {:a 1 :b 2}) ;; ([:b 2] [:a 1])
\end{verbatim}

In the subsequent code, we are forced to put the result in its own list or vector in this case. This will add the result to the end (or right-hand side) of the running result list.

\begin{verbatim}
;; What we want
(lazy-cat '(1)
   (merge
    (zipmap etal-keys
       (map #(% {:last-trade-price 6.111604269033163, :last-trade-time trade-time})
        etal-keys))
    {:last-trade-price-average 6.0972436})))

;; (1
Developing the Simple Moving Average

We now properly append our average results to the right-hand side of the list, which corresponds to how the input was originally partitioned. To call simple-moving-average, we'll operate on our timeseries sequence, which was defined in the Reasoning about the equation needed to achieve our output section. Remember that the first average represents the first 20 price points, thus corresponding to the 20th item in our price list. The following algorithms will continue along this line:

```
  (simple-moving-average {} 20 timeseries)
  ([:last-trade-price {:last 5.466160487301605},
    :last-trade-time #inst "2015-09-24T04:13:13.868-00:00",
    :last-trade-price-average 7.194490217405031,
    :population
      ([:last-trade-time #inst "2015-09-24T04:13:13.868-00:00",
        :last-trade-price {:last 5.466160487301605}]
      
      [:last-trade-time #inst "2015-09-24T04:13:15.868-00:00",
        :last-trade-price {:last 6.540895364039775}]
      ...
      )
  )
  [:last-trade-price {:last 6.540895364039775},
   :last-trade-time #inst "2015-09-24T04:13:15.868-00:00",
   :last-trade-price-average 7.180900526986235,
   :population
     ([:last-trade-time #inst "2015-09-24T04:13:15.868-00:00",
       :last-trade-price {:last 6.540895364039775}]
      
      [:last-trade-time #inst "2015-09-24T04:13:16.868-00:00",
        :last-trade-price {:last 5.53301182972796}]
      ...
      )
  ...
```

Destructuring

Earlier, I gave an explanation of a destructuring code block. Here, we'll elaborate on it a little further. The inner let block is where we assigned values to the input-key, output-key, and etal-keys. We can assign these symbols for later use as keys in maps. The input-key, which is assigned in the let block, is later used to pull out the last price from our input list. The output-key is the last average that we calculated in our function. The etal-keys are vectors of price and time, which we used to join those keys with previous values, thereby adding the average key (or output key) and value to each one:

```
(let [  ...
   {input-key :input
The syntax, as we introduced in the previous chapter, uses something called **destructuring**, which is a small custom language [or little language or domain-specific Language (DSL)] in Clojure. This is another example of the possibilities offered by macros and code rewriting. Destructuring lets you extract values from data structures and bind them to local symbols (symbols are explained in Chapter 1, Orientation – Addressing the Questions Clojure Answers). It works in let bindings, function parameters, or macros that expand to one of these forms. You can extract symbols from vectors or maps. Refer to the official documentation at [http://clojure.org/vars](http://clojure.org/vars) for a more exhaustive breakdown. In our case, we want to pull out Vars from a map and allow for default values if a user doesn't supply any of the required entries. So, let's say we supply the `{:input :fubar :output :thing :etal [:one :two]}` map to the destructuring form. The first set of mappings will be applied and we'll see a result like this:

```clojure
(let [{input-key :input
       output-key :output
       etal-keys :etal
       :or {input-key :last-trade-price
            output-key :last-trade-price-average
            etal-keys [:last-trade-price :last-trade-time]}}
      {:input :fubar
       :output :thing
       :etal [:one :two]}])
```

```clojure
(println input-key)
(println output-key)
(println etal-keys))
```

```clojure
;; :fubar
;; :thing
;; [:one :two]
```

However, if we give a nil value to our destructuring form, the default bindings after :or, will be applied. We'll then get a result like this:

```clojure
(let [{input-key :input
       output-key :output
       etal-keys :etal
       :or {input-key :last-trade-price
            output-key :last-trade-price-average
            etal-keys [:last-trade-price :last-trade-time]}}
      options]
  ...)
```

```clojure
;; :input :output
;; :etal
;; [:last-trade-price :last-trade-time]
```
Developing the Simple Moving Average

etal-keys :etal
 :or {input-key :last-trade-price
     output-key :last-trade-price-exponential
     etal-keys [:last-trade-price :last-trade-time]}
   nil}

  (println input-key)
  (println output-key)
  (println etal-keys))

;; :last-trade-price
;; :last-trade-price-exponential
;; [:last-trade-price :last-trade-time]

Here's a sample graph output (your data may vary):

![Graph Output](image-url)
Summary

In this chapter, we built on the simple-moving-average function using the more advanced partition Clojure data transformation function. Vars, binding, and lazy sequences helped us organize our expressions. And our function design was guided by the data input and desired data output. The goal here is to form a solid understanding of a problem, easily represent data, and quickly perform calculations. From our basic tick list, we were able to (from a starting point) calculate the average price at each point up to the current price. We did this by slicing or partitioning our list so that each price point had a history of the previous 20 price points. These are the kinds of functions that will empower us to manipulate and shape our data. We'll take the lessons we've learned in this chapter to the next chapter and implement a slightly more advanced algorithm—the exponential moving average.
Where to buy this book

You can buy Clojure for Finance from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.