Community Experience Distilled

Clojure for Finance

Leverage the power and flexibility of the Clojure language for finance

Timothy Washington

[PACKT] open source*
PUBLISHING
community experience distilled

Clojure for Finance

Leverage the power and flexibility of the Clojure language for finance

Timothy Washington

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Clojure for Finance

First published: January 2016

Production reference: 1040116

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-928-6

www.packtpub.com

Credits

Author
Timothy Washington

Reviewers
Ed Babcock
Nicholas Quirk
Dajana Štiberová

Acquisition Editors
Richard Brookes-Bland
Divya Poojari

Content Development Editor
Shweta Pant

Technical Editor
Rahul C. Shah

Copy Editor
Sonia Cheema

Project Coordinator
Sanjeet Rao

Proofreader
Safis Editing

Indexer
Hemangini Bari

Graphics
Disha Haria

Production Coordinator
Nilesh Mohite

Cover Work
Nilesh Mohite

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.

Thanks to Richard Brooks-Bland for first convincing me to write this book. This wouldn't have happened without your goading. I'd also like to thank Chris Zheng for the first bits of feedback, and all the reviewers, for their hard work, catching errors I would have otherwise missed.

About the Reviewers

Ed Babcock is a software developer who has been enjoying Clojure and ClojureScript since first hearing about them back in 2011. Over the last few years, he has been able to use Clojure for a variety of projects, ranging from mobile and web development to a large-scale search engine. Having worked with other programming languages in the past, he is certain that Clojure's simple design patterns and excellent community have made him a much better programmer. He is currently working for HomeSpotter, a home search start-up, and exploring cross-platform application development though several open source projects on GitHub.

Nicholas Quirk is currently finishing his master's degree in computer science. He has 5 years of industry experience, primarily with Java web application development; however, he also loves LISP dialects, such as Clojure, and the history of computer science itself. His staples in life are reading and programming. If you can't find him doing either of these, he might be doodling, hiking, mountain stomping, or enjoying the company of friends and family.

> I would like to thank my friends and family for their support as I've bought my first house.

Dajana Štiberová is a young Clojure enthusiast who's based in Bratislava, Slovakia. After deciding to abandon her work in the finance sector, she rediscovered her passion for math and analytical thinking with the help of programming. She currently works on distributed testing platforms that are written almost entirely in Clojure.

She likes to spend her free time cycling to discover new roads and places, helping animals, and seeks to improve the lives of children in hospitals.

Table of Contents

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 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.

What you need for this book

You just need Leiningen and one of the common OSes, including OSX, Linux, Windows, and that's it!

Other than this, you need the following hardware:

- MacBook Pro (Retina, Mid 2012)
- A 2.3 GHz Intel Core i7 processor
- 8 GB 1600 MHz DDR3 RAM

You also need the following software:

- OSX Yosemite (10.10.2)
- Java version "1.8.0_45"
- Leiningen 2.5.2
- Clojure 1.7.0

Who this book is for

If you're a finance professional who is currently using VBA (Excel) to perform financial calculations and quantitative analysis, and would like to use Clojure to augment your existing toolset, then this book is for you.

Basic knowledge of financial concepts is essential. Basic programming knowledge would also be an added advantage.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can test this out by running `java -version` in the shell or command prompt provided by your system."

A block of code is set as follows:

```
(defn generate-prices [lower-bound upper-bound]
      (filter (fn [x] (>= x lower-bound))
              (repeatedly (fn [] (rand upper-bound)))))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
(defn generate-sine-sequence []

  (let [ein (randomize-vertical-dilation sine 0.5 2.7)
        zwei (randomize-horizontal-dilation ein 0.3 2.7)
        sine-partial (partial zwei 0)

        xinterc-sine-left (find-xintercept - sine-xintercept)
        xinterc-sine-right (find-xintercept + sine-xintercept)

        granularityS (rand-double-in-range 0.1 1)
       xsequenceS (iterate (partial + granularityS) xinterc-sine-left)]

    (map sine-partial xsequenceS)))

(defn generate-oscillating-sequence []
  (four analytics/generate-prices-without-population 5 15))
```

Any command-line input or output is written as follows:

```
> (alter-var-root #'system component/start)
> (clojure.pprint/pprint system)
{:tms
 {:channel
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]},
 :cns
 {:timeseries
  {:channel
   #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]},
  :channel
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]}}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To be precise, these values are called **Vars**."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Orientation – Addressing the Questions Clojure Answers

Hello. In this book, I'll try to demonstrate Clojure's utility in processing and analyzing financial data. Whether it is core banking, risk assessment, compliance, and so on, this book's audience mostly comprises finance professionals who use Clojure to improve their Excel spreadsheets or existing toolsets. As such, this will be a beginners programming book, but it won't be light on substance. I want to assert a broader philosophical point: better tools make your professional life easier. By better, I mean tools that offers greater expressive power, which is a measure of a tool's capacity to let you do and see more with the same or less amount of effort as compared to other tools.

To justify replacing or augmenting your existing tools, I'll try to point out the benefits of using functional programming (Clojure, in this case) in an organization in order to fully represent and manage capital. You might want to know how you can quickly and effectively analyze financial conditions and decide when and how to invest and maximize capital growth. So, I'll begin by weaving together the two concepts of capital flows and state-of-the-art computation. You will see how concepts, such as **functional programming** (**FP**), part of computer science's cutting edge, enable a high fidelity of data representation and transformation and information extraction.

The goals of this book are as follows:

- Highlighting the importance of quickly and effectively representing data and its calculations
- Showing how Clojure can easily represent data and quickly perform calculations
- Giving you confidence to deal with basic data types and manipulations

- Pointing you to more advanced approaches to calculate and transform data as well as build algorithms

There are an unlimited number of financial equations we can study (such as the cost of capital, rate calculation, fee calculation, and so on). So, I think it will be more useful to focus on one particular application. As such, this book's pedagogy will be to walk you through slowly building a set of lagging price indicators that follow a moving stock price time series. I'll also introduce you to topics, such as I/O, concurrency, macros, types, and type theory. However, as these topics are out of scope for this book, we won't go into very much detail about them. These are tools that are used in building software libraries and systems. There are several places that you can refer to for more advanced information.

Notions of computation

I'll state up front my position that computer programming languages are simply tools that let us communicate with other programmers and give human control of the machine. But, what does it mean to compute? Alan Turing was an English mathematician who invented the disciplines of computer science and artificial intelligence. You can read more about him at `http://en.wikipedia.org/wiki/Alan_Turing`. He believed that computation can be thought of as an actualization or concrete version of a mathematical function. His ideas of computation can be found in the contributions he made to the concept of an algorithm and his description of a Turing Machine. He perceived an algorithm to be a step-by-step procedure used for calculations. These steps are expressed as a finite list of well-defined instructions to calculate a function. Further, Turing postulated a theoretical computer (the Turing Machine) that could calculate anything that is computable, in spite of the complexity involved.

Others, such as Alonzo Church, approached the notion of computation by trying to solve the Entscheidungsproblem, which is German for a decision problem (`http://en.wikipedia.org/wiki/Entscheidungsproblem`). The decision problem asks for an algorithm, where the input is a statement of first-order logic and answers yes or no according to whether the statement is universally valid. Alonzo Church proposed the lambda calculus as a formal system in mathematical logic that satisfied this decision problem. The lambda calculus expresses computation based on function abstraction and application using variable binding and substitution (`http://en.wikipedia.org/wiki/Lambda_calculus`).

Further still, MIT professor, Gerald Sussman, says, "We (as humans) really don't know how to compute" (you can read more at `http://www.infoq.com/presentations/We-Really-Dont-Know-How-To-Compute`). That is in comparison with data and computation that we see in the natural world. Plants, for example, can harvest up to 95% of energy from the light that they absorb. They transform sunlight into carbohydrates in one million billionths of a second, preventing much of this energy from dissipating as heat. This near instantaneous process uses the basic principle of quantum computing—the exploration of a multiplicity of different answers at the same time—to achieve near perfect efficiency. The protein structure of a plant, which can be derived using this quantum effect, somehow allows the transfer of energy but not heat. According to David Biello, "Inside every spring leaf is a system capable of performing a speedy and efficient quantum computation" (you can read more about it at `http://www.scientificamerican.com/article/when-it-comes-to-photosynthesis-plants-perform-quantum-computation/`). We as humans just don't understand how it happens. Gerald Jay Sussman compares our computational skills with a genome, concluding that we are way behind in creating complex systems such as living organisms.

Even if our tools and approaches are primitive, we can still grasp the importance of such capabilities. The structures of the modern world dictate tools that let humans quickly calculate, abstract, and derive algorithms (ways of representing abstract data, credit in this case). The computation tools we use simply give us fine-grained control of these problems. Without them, the modern world as it is today would not be possible. Strong computation has applications in every real-world discipline [engineering, science, business (credit and financial systems), and so on], or at least any discipline where data can be represented digitally.

Notions of finance

It's useful to think of capital as a store of time and labor. Let's consider money outside our current banking system. In The Ascent Of Money: A Financial History of the World by the Penguin Group, Niall Ferguson discusses the history of money and credit in human civilization (you can find it at `http://www.amazon.com/The-Ascent-Money-Financial-History/dp/0143116177`). This approach lets us consider how societies use capital to distribute wealth. Among other things, he argues that the evolution of credit and debt is as important as any technological innovation in the rise of a civilization. I've mentioned this to highlight the purpose of finance and the fact that money and finance have existed across many banking systems, including before our current one. It's important that this approach not be ideological. I want to frame the core purpose and function of a bank and flesh out the mechanics of money flows.

When programming (computers), the first task is to build an understanding of the systems we're trying to model and the problems that we are trying to solve. As such, let's distill some central notions of money and how it fits into the banking function. This is written in the spirit of banks simply becoming information processes and looking more like software companies. Technology produces alternative mediums of capital, such as cryptocurrencies, for example, bitcoin (you can find more about this at `http://en.wikipedia.org/wiki/Cryptocurrency`). But this concept is outside the scope of the book. I make the point to emphasize the seismic effect that software is having on finance and all modern professions. I will try to describe the context and constraints within which such banking software must operate.

Money is a medium of exchange, unit of account (divisible, fungible, or of a specific measure or size), store of value, and a standard of deferred payment. In economics, money creation is the process by which the money supply of a country or a monetary region (for example, the EU) is increased. Changes in the quantity of money may be caused due to the actions of a central bank, depository institutions (principally commercial banks), or the public. However, the major control rests with a central bank. The actual process of money creation takes place primarily in banks (refer to `https://archive.org/details/ModernMoneyMechanics` and `http://upload.wikimedia.org/wikipedia/commons/4/4a/Modern_Money_Mechanics.pdf`). In the U.S., the money creation process centers around the US Federal Reserve. So, a central bank may introduce new money into an economy by purchasing financial assets or lending money to financial institutions or governments. The majority of money in our modern economy is created by commercial banks who give loans or demand deposits.

There's much more to take into account. However, this context allows us to begin considering how to faithfully represent credit levels and flows within and between bank and bank-like entities. We can also start thinking about a commercial bank's core functions and constraints, and the systems it should implement to perform all of these.

Concrete types of computation

Most to all computer languages are Turing Complete or can compute every Turing-computable function. This refers to a language that has conditional branching and allows an arbitrary number of variables. If this is the case, then how do different language categories address the notion of computation?

- **Imperative or procedural programming (Assembly, C, Fortran, et al.)**: This generally describes computation in terms of statements that change a program state.

- **Object-Oriented programming (Simula, Smalltalk, Java, C++, et al.)**: This grew out of a need to build larger and more complex systems. Its computation model focuses on data encapsulation and object interaction. While procedural programming treats computation in terms of statements that change a program state, the **object-oriented** (**OO**) paradigm encapsulates program states in objects and procedures in methods.
- **Functional programming (Lisp , R, Haskell, APL, et al.)**: This treats computation as the evaluation of mathematical functions and avoids state and mutable data. In contrast, procedural and object-oriented programming emphasize changes in state, while Functional Programming FP emphasizes the application of functions, very often with immutable data. FP has its roots in the lambda calculus, and many functional programming languages can be viewed as elaborations of lambda calculus.

There are other categories and techniques being researched, such as logic programming, type theory, and so on. We'll focus mainly on functional programming and Clojure, as they provide sufficient and powerful expressivity needed to model our problem domains. These provide a high fidelity of information, faithfully representing credit levels and credit flows within and between bank and bank-like entities. For example, the notion of a function used in imperative programming is that it can have side effects that may change the value of the program state. Functions lack referential transparency, where the same language expression can result in different values at different times depending on the executing program state. This has negative implications when making financial calculations or representing credit properly in your systems. Conversely, in FP, the output value of a function depends only on the arguments that are input to the function, which is much closer to a mathematical function. Eliminating these side effects make it much easier to understand and predict the behavior of a program. This and FP's close proximity to mathematical functions are some of the key motivations for the development of functional programming.

Tooling

Let's begin by setting up the tools we'll need to run the book's code. You can author the code for this book in any text editor you prefer. I'll be using Emacs. Leiningen will be the build tool that we'll use to compile and run our Clojure code. It lets us leverage Clojure's many key features shown as follows:

- Functional programming or a first class function evaluation model (a la lambda calculus).
- Immutability is a feature where data that is created is never changed. It is only transformed via the functions you apply.

- Laziness is the feature of not evaluating any code until it is absolutely required. This saves computing resources.
- Homoiconic, which refers to the programs you write, is actual data and vice versa.
- A well-designed, which syntax makes your code easy to read and reason out.
- A **Read-Eval-Print-Loop** (**REPL**) lets us quickly evaluate our code and make changes in real time.

The following steps will guide you to create your first Clojure project:

1. The first requirement is a Java runtime, if you don't already have one. You can test this out by running `java -version` in the shell or command prompt provided by your system. If you don't have Java, you can download it from `https://java.com/en/download/`. You can also consult your OS' software or package management tool for the best way to install Java.
2. The next step is to install Leiningen (you can do this from `http://www.leiningen.org`). Follow Leiningen's installation instructions by visiting `http://leiningen.org/#install`. You can test your installation by executing `lein` to see the help menu that is produced.
3. Now we're ready to create a Clojure project and perform some simple operations.
4. In your shell, within the directory of your choice, run `lein new edgar`. There should be a new project directory there named `edgar/`.
5. Next, change into the `edgar/` directory and run `lein repl`. After a few seconds, you will be brought into an REPL.
6. From here, you can begin to evaluate primitive expressions, such as numbers and strings. In your REPL, type `1` [*Enter*], and then `"foobar"` [*Enter*].
7. Clojure is also known as a list processing language or a LISP. Now we're going to evaluate a list with a function as the first argument, which is `(+ 1 1)`.
8. Clojure functions are evaluated in the first position of a list with all function arguments, including subexpressions, placed afterwards.
9. Try evaluating `(+ 5 (* 2 4 6))`. Here, the `*` function is applied to `2 4 6`, which produces `48`. After this, the REPL applies `+` to `5` and `48`, yielding `53`.

Clojure has some special reserved characters that it needs to use exclusively. As such, we won't be able to use them when naming our program elements. The Macro characters section of Clojure's Reader page at `http://clojure.org/reader`, itemizes them.

A first look at Clojure's core functions

There are many places you can refer to for Clojure's core functions such as the official Clojure documentation (`http://clojure.org/documentation`). Additionally, this site provides a handy cheatsheet that gives a bird's-eye view of core functions and their categories (`http://clojure.org/cheatsheet`).

These are a few community maintained sites that help you get an overview of Clojure's core functions:

- `https://clojuredocs.org/quickref` (ClojureDocs)
- `http://conj.io` (Grimoire)
- `http://clojure-doc.org` (Clojure Documentation; source: `https://github.com/clojuredocs/guides`)

Let's take a look at the Grimoire Community Clojure documentation. If you navigate to `http://conj.io`, you'll see a good mix of the official cheatsheet's overview, with a dynamic ability to filter on function names that we type into a search field. The functions are also grouped into overriding concerns. We'll only touch on a few functions from each category. The following section will provide you with a Clojure environment where you can explore Clojure's core functions.

[Note that the text following `;;` refers to comments and is not evaluated by Clojure.]

Primitives

Let's have a look at the following list of primitives:

- **Nil**: This simply represents the absence of a value. It is the lowest in any sort order (for example, when appearing in a sorted set) and equates to a logical false (for example, when appearing in an `if` condition):

  ```
  nil  ;; nil
  (type nil)  ;; nil
  ```

- **Booleans**: Clojure's primitive boolean types are `true` or `false`. These are java `java.lang.Boolean` classes under the hood:

  ```
  true  ;; true
  (type false)  ;; java.lang.Boolean
  ```

- **Numbers**: These are functions to help you test, generate, and manipulate numbers. Here is an example of how to apply the most basic number functions:

  ```
  (+ 9 5)  ;; 14
  (- 400 18.75)  ;; 381.25
  (* 6 10)  ;; 60
  (/ 50 5)  ;; 10
  (quot 50 5)  ;; 10 - the quotient (same as division `/`) of
  the numerator and denominator, respectively
  (rem 26 12)  ;; 2 - the remainder of the numerator and
  denominator, respectively
  (mod 26 12)  ;; 2 - the modulus (same as remainder `rem`)
  of the numerator and denominator, respectively
  (inc 67)  ;; 68 - increment the argument (a number) by 1
  (dec 100)  ;; 99 - decrement the argument (a number) by 1
  (max 250 45)  ;; 250 - the maximum number between the first
  and second arguments
  (min 250 45)  ;; 45 - the minimum number between the first
  and second arguments
  ```

- **Characters**: Characters in Clojure are either single nonnumeric (`*`, `+`, `!`) or alphanumeric (`a`, `s`, `d`, `f`) things. Characters are prefixed with a backslash `\h, \e, \l, \l, \o`. We can also represent special characters, such as a `\newline`, `\space`, `\return`, and so on.
- **Strings**: Strings are a simple sequence of characters. They are surrounded by double quotes, such as `"hello"`. As you'll see in the next section, this is very close to the definition of a collection. So, some collection functions work on strings as well. Any function that takes a collection and doesn't require it to be a persistent collection can often operate on a string:

  ```
  (reverse "hello")  ;; (\o \l \l \e \h)
  (count "hello")  ;; 5
  (first "hello")  ;; \h
  ```

  This is a subtle distinction. Sometimes, you'll need to pull in string-specific functions that do what you mean. The `clojure.string/reverse` function versus the `clojure.core/reverse` function is such an example:

  ```
  (require '[clojure.string :as s])

  (s/reverse "hello")  ;; "olleh"
  (reverse "hello")  ;; (\o \l \l \e \h)
  (s/join ", " "hello")  ;; "h, e, l, l, o"
  ```

We can also apply functions to patterns of strings. These patterns are described with regular expressions, which are written as strings (double-quotes) and prefixed with a hash. Regular expressions follow Java's Regular Expression system, which is specified in the `java.util.regex.Pattern` class, available at `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`. They can often be a more expressive way of targeting a string pattern in a large block of text. For example, we can target only continuous blocks of numbers. Or, we can target a specific word or character sequence, as shown here:

```
(re-find #"\d+" "hello1234")  ;; "1234"
(s/replace "Hello World" #"World" "Clojure")  ;; "Hello
 Clojure"
```

- **Symbols**: Symbols in Clojure point to things in a system runtime. For example, the core functions we have looked at so far are actually represented by symbols that point to a function. So, if we try to evaluate `inc` or `first` outside a list, the runtime system will return a gnarly looking function representation. We can take this result function and use it somewhere else. Symbols can contain any alphanumeric character, must begin with a nonnumeric character, and can also contain `*`, `+`, `!`, `-`, `_`, and `?`. We can create our own symbols or use a function, such as `def`, to assign values to symbols:

```
(symbol 'hello)  ;; hello
(symbol "hello")  ;; hello
(symbol "clojure.core" "hello")  ;; clojure.core/hello
(def hello "world")  ;; #'user/hello
hello ;; "world"
```

- **Keywords**: Keywords are like symbols, except they are self-referencing. If you pass a keyword to a Clojure environment, you'll get back the same `:hello  ;; :hello` value. This makes them handy as keys or identifiers to other associations. For example, a popular Clojure pattern is to use keywords as keys in maps (discussed shortly) or other associative datatypes. This is because keywords act as self-referring functions. Being able to act like functions means that we can use keywords as functions of other data structures. So taking the previous example of the map, we can look up values on a map, such as `(:foo {:foo "bar" :hello "world"})`, and get the result, which is "`bar`":

```
:hello  ;; :hello
(:foo {:foo "bar" :hello "world"})  ;; "bar"
```

Collections

All of these Clojure collection types are what is known as immutable. This property means that once a collection has been created, it cannot change. If we want to operate on a collection, we can call a function over it. A new collection is returned and it shares the original's structure plus any modifications resulting from the function. This, as we'll explore later on, is a central part of Clojure's computation model. And, it turns out that it's an efficient use of a computer's resources. Let's take a look at the following Collection types:

- **Lists**: Lisps, including Clojure, have a syntax that is represented directly in data structures. This is what's known as homoiconic—where the code is the data. We'll explore this concept later on. However, homoiconicity offers a lot of benefits, including the ability to dynamically change your running program or even produce new programs if you want. This is an advanced concept that we'll explore a little more in later chapters. Lists in Clojure (or any Lisp) are its most rudimentary data structure. They are a linear sequence of items. The first item points to the next and so on until the end. They are also the primary way of invoking functions. Evaluating this following form will apply the `max` function to its two arguments `(max 4 90)`. In this format, the function always goes at the beginning of the list. All arguments to this function come after. An unevaluated list is only a data structure. As such, we can put anything at the beginning of a list (not just a function). However, we must prefix lists with an apostrophe (called quoting), for example, `'(4 90 "hello")`. Another important fact to note is that the efficient point of access for a list is its head. Randomly accessing a list, say, in its middle, is relatively inefficient:

  ```
  (conj '("a" "s" "d" "f") "z")  ;; ("z" "a" "s" "d" "f")
  ```

  Here, `"z"` gets "conjoined" to the front of the (`"z" "a" "s" "d" "f"`) list, due to list's head first efficiency attribute

  ```
  (count '("a" "s" "d" "f"))  ;; 4
  (empty '("a" "s" "d" "f"))  ;; ()
  (empty? '("a" "s" "d" "f"))  ;; false
  ```

- **Vectors**: Vectors are also a linear sequence of items from the beginning to the end, as shown in the following code. They are denoted by opposing square brackets, such as `[75 6 452 40]`. The difference, as compared to a list, is that they give efficient random access. They are also meant to provide a visual break from lists. This makes them comparatively easier than other Lisps for the purpose of reading Clojure source code. A vector is closer to an array as opposed to a list, which is more like a linked list:

  ```
  (conj ["a" "s" "d" "f"] "z")  ;; ["a" "s" "d" "f" "z"]
  ```

Here, `"z"` gets conjoined to the end of the vector as vectors offer efficient random access

```
(mapv inc [1 2 3 4])  ;; [2 3 4 5]
```

This applies (or maps) the `inc` function over each of the vector's elements

- **Sets**: Sets are a collection of unique items. This is to say that we can only put one of each item into a set. They are denoted by curly braces, prefixed with a hash, for example, `#{1 2 3 4}`. Your Clojure environment will give you an error if you try to create a literal set with a duplicate item, which is `#{1 2 3 3}`. There's also a neat trick to note. Sets (along with maps and keywords) can act as functions. In this case, a set can be placed in the first position of a list and then applied to an argument. What's happening here is that the set is acting as the "does this exist" function of its items. So, `(#{"a" "s" "d" "f"} 2)` will return `nil`, because `2` is not in the set. However, `(#{"a" "s" "d" "f"} "d")` will return the `d` item as it is in the set:

  ```
  (#{"a" "s" "d" "f"} 2)  ;; nil
  (#{"a" "s" "d" "f"} "d")  ;; "d"
  ```

- **Maps**: Maps are Clojure's associative data structure. They associate keys to values. Keys and values can include any Clojure data structure. But, very often, you'll see Clojure code using keywords as keys. Maps are denoted by opposing curly braces, for example, `{:hello "world" :foo "bar"}`. Similar to sets, maps act as functions of their elements—in the form of an associative look up in this case. So, if I evaluate `({:hello "world" :foo "bar"} :foo)` in a Clojure runtime environment, I'll get back `bar`:

  ```
  ({:hello "world" :foo "bar"} :foo)  ;; "bar"
  (assoc {} :a "b")  ;; {:a "b"}
  (zipmap [:hello :foo] ["world" "bar"])  ;; {:foo "bar", :hello "world"}
  ```

There are many more core Clojure functions. Some of the categories you should be seeing in Grimoire are mentioned as follows. Further in the book, we'll explore the concepts of vars, macros, first-class functions, sequences, and so on. But at least for now, you have a place that you can go to look up functions that pertain to these topics:

- **Sequences**: Sequences are a central part of Clojure's computation model. They are a logical sequential representations over a more concrete form, such as a vector, list, set, and so on. Most of Clojure's core functions (`map`, `reduce`, and so on) operate on and produce sequences.

- **Functions**: Functions in Clojure form the core of how to do things. As in other functional programming languages, functions are first class program elements that can be passed around and transformed just like data. As such, Clojure has functions that can change functions to delay evaluation, change their argument order, and so on.
- **Macros**: Any computer language's syntax gets translated to an internally abstract tree in order to be run. Since Clojure code is already data (the property of being homoiconic), macros provide a way of changing an actual program.
- **Reader Macros**: These are preexisting macros that are available in the language's runtime. They perform useful tasks such as converting a string into an internal date representation.
- **Metadata**: This contains data about our functions or program elements. Metadata is attached directly to the thing it describes. However, it does not factor in the actual value of its subject. It simply provides useful information, such author information, documentation, and so on.

There are many more categories of functions to explore. Having a look at the various functions in Grimoire and the official documentation, will begin to give you a feel for each's category, and, thus, utility.

Summary

In this chapter, we were introduced to the notions of computation and finance. This gave us the background required to understand Clojure's utility to solve real-world problems in many domains and finance in particular. It also gave us an orientation to help us understand Clojure's approach to computation. Clojure fits under the banner of functional programming, a different approach from imperative or object-oriented programming. We took a first look at its core Functions that act on primitives (numbers, characters, strings, symbols, and keywords) and Collections (lists, vectors, sets, and maps). Many other core functions and their categories (sequences, functions, and others) will be addressed further on in this book. Finally, we installed the Leiningen build and run tool, which will help us build and run our functions throughout the book.

In the next chapter, we'll take a look at Clojure's principles and core artifacts. We'll begin to apply these to our financial data structures and functions.

2
First Principles and a Useful Way to Think

We looked at central notions of computation and finance in the last chapter so that we could come up with the best approach to represent our financial data. In essence, you're learning how to fish. We're going to build on this knowledge right away and start developing an equation to generate price data. At each step, we're going to look more closely at core functional programming features. This should give you a solid understanding of important concepts in their immediate context.

In this chapter, we will cover the following topics:

- Function evaluation
- First-class functions
- Lazy evaluation
- Basic Clojure functions and immutability
- Namespaces and creating our first function
- The Read-Eval-Print-Loop (REPL)
- Basic data structures
- Macros and more in-depth data transformation

Let's begin by looking at the data we need to represent.

Modeling stock price activity

There are many types of banks. Commercial entities (large stores, parking areas, hotels, and others) that collect and retain credit card information are either quasi banks or farm out credit operations to bank-like processing companies. There are other well-known consumer banks, which accept demand deposits from the public, and a range of other banks, such as commercial banks, insurance companies and trusts, credit unions, and in our case, investment banks. As promised, this book will slowly build a set of lagging price indicators that follow a moving stock price time series. In order to do this, I think it's useful to touch on stock markets and crudely model stock price activity. A stock (or equity) market is a collection of buyers and sellers trading economic assets, usually companies (for more details on the stock market, visit `http://en.wikipedia.org/wiki/Stock_market`). The stock (or shares) of these companies can be equities that are listed on an exchange (the New York Stock Exchange, London Stock Exchange, and so on) or traded privately.

Earlier, we reviewed the nature of capital. This helped to frame the core purpose and function of a bank and flesh out the mechanics of money flows. From here on, we can create an imaginary investment bank. Chartering a bank would also involve plugging it into an imaginary central bank network. Once this is done, our bank would need to register to trade on one or more exchanges. Now, our bank can (among other things) buy and sell on a stock market.

Our next exercise will do the following:

- Crudely model stock price movement, giving us a test bed to write our lagging price indicators
- Introduce some basic features of the Clojure language

Function evaluation

The Clojure website has a cheatsheet (`http://clojure.org/cheatsheet`) with all the language's core functions. The first function we'll look at is `rand`, a function that randomly gives you a number within a given range. So, in your `edgar/` project, launch the **Read-Eval-Print-Loop** (**REPL**) with the `lein repl` shell command. After a few seconds, you will be brought into a REPL. Again, Clojure functions are executed by being placed in the first position of a list. The function's arguments are placed directly afterwards as follows:

- In your REPL, evaluate `(rand 35)`, `(rand 99)`, `(rand 43.21)`, or any number you fancy

- Run it many times to see whether you can get a different floating point number within 0 and the upper bound of the number you've provided

First-class functions

The next functions that we'll look at are `repeatedly` and `fn`. The `repeatedly` function takes a second function and returns an infinite (or *n* length if supplied) lazy sequence of calls to this second function. This is the first encounter with a function that can take another function. We'll also encounter functions that return other functions. Described as first-class functions, these fall out of the lambda calculus and are the central features of functional programming.

As such, we need to wrap our previous `(rand 35)` call in another function. The `fn` function is one of Clojure's core functions and produces an anonymous, unnamed function. We can now supply this function to `repeatedly`.

Lazy evaluation

We have only taken the first 25 numbers of the `(repeatedly (fn [] (rand 35)))` result list because the list (actually a lazy sequence) is infinite. Lazy evaluation (or laziness) is a common feature in functional programming languages, `http://en.wikipedia.org/wiki/Lazy_evaluation`. Being infinite, Clojure chooses to delay evaluating most of the list until it's needed by some other function that pulls out some values. Laziness benefits us by increasing performance and letting us easily construct control flow. We can avoid needless calculation, repeated evaluations, and potential error conditions in compound expressions. Let's try to pull out some values with the `take` function. The `take` function itself returns another lazy sequence of the first *n* items of a collection.

Evaluating `(take 25 (repeatedly (fn [] (rand 35))))` will pull out the first 25 `repeatedly` calls to `rand`, which generates a float between 0 and 35.

Basic Clojure functions and immutability

There are many operations we can perform over our result list (or lazy sequence). One of the main approaches of **functional programming** (**FP**) is to take a data structure and perform operations over it to produce a new data structure or atomic result (such as a string, number, and so on). This may sound inefficient at first. However, most FP languages employ something called **immutability** to make these operations efficient. Immutable data structures are those that cannot change once they've been created. This is feasible as most immutable FP languages use some kind of structural data sharing between an original and a modified version. The idea is that if we run evaluate `(conj [1 2 3] 4)`, the resulting `[1 2 3 4]` vector shares the original vector of `[1 2 3]`. The only additional resource that's assigned is for any novelty that's been introduced in the data structure (`4`). There's a more detailed explanation of, for example, Clojure's persistent vectors here: `http://hypirion.com/musings/understanding-persistent-vector-pt-1`:

- `conj`: This conjoins an element to a collection—the collection decides where. So, conjoining an element to a `(conj [1 2 3] 4)` vector versus conjoining an element to a `(conj '(1 2 3) 4)` list yields different results. Try doing this in your REPL.
- `map`: This passes a function over one or many lists, yielding another list. Evaluating `(map inc [1 2 3])` increments each element by 1.
- `reduce` (or left fold): This passes a function over each element, accumulating one result. Evaluating `(reduce + (take 100 (repeatedly (fn [] (rand 35)))))` sums the list.
- `filter`: This constrains an input by providing some condition.
- `>=`: This is a conditional function that tests whether the first argument is greater than, or equal to the second function. Try `(>= 4 9) and (>= 9 1)`.
- `fn`: This is a function that creates a function. This unnamed or anonymous function can have any instructions that you choose to put in it.

So, if we only want numbers above 12, we can add this assertion to a predicate function. Try entering the following expression into your REPL:

```
(take 25 (filter (fn [x] (>= x 12))
                 (repeatedly (fn [] (rand 35)))))
```

Namespaces and creating our first function

We now have the basis to create a function. It will return a lazy infinite sequence of floating point numbers within an upper and lower bound. `defn` is a Clojure function that takes an anonymous function and binds a name to it in a given namespace. A Clojure `namespace` is an organizational tool used to map human readable names to things, such as functions, named data structures, and so on. Here, we're going to bind our function to the `generate-prices` name in our current namespace. You'll notice that our function is starting to span multiple lines. This will be a good time to author the code in your text editor of choice. I'll be using Emacs (you can read more about Emacs at `https://en.wikipedia.org/wiki/Emacs`):

1. Open your text editor and add this code to the file called `src/edgar/core.clj`. Make sure that `(ns edgar.core)` is at the top of this file.
2. After adding the following code, you can then reload the code in your REPL. `(load "edgaru/core")` uses the `load` function to load the Clojure code in `src/edgaru/core.clj`:

```
(defn generate-prices [lower-bound upper-bound]
    (filter (fn [x] (>= x lower-bound))
            (repeatedly (fn [] (rand upper-bound)))))
```

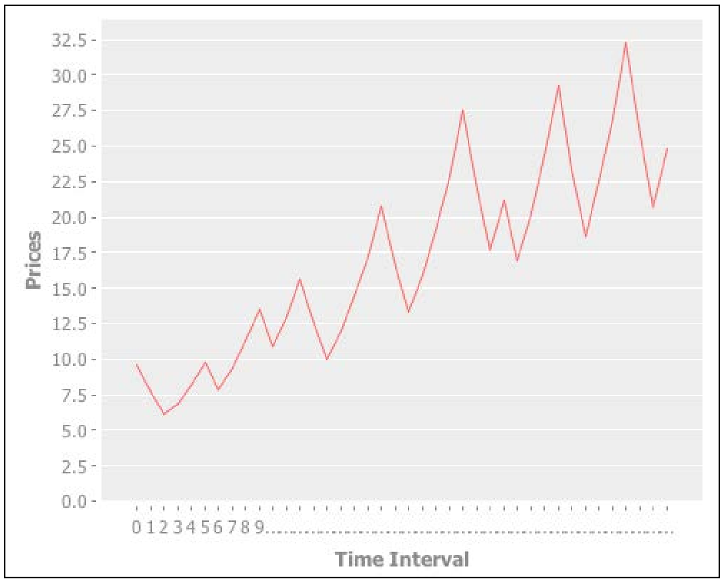
The Read-Eval-Print-Loop

In our REPL, we can pull in code in various namespaces with the help of the `require` function. This applies to the `src/edgar/core.clj` file we've just edited. Here is the code in the `edgar.core` namespace:

1. In your REPL, evaluate `(require '[edgar.core :as c]);` `c` is just a handy alias we can use instead of the long name.
2. You can then generate random prices within an upper and lower bound. Take the first 10 of them `(take 10 (c/generate-prices 12 35)).`
3. You should see results akin to the following output. All elements should be within the range of 12 to 35:

```
(29.60706184716407 12.507593971664075 19.79939384292759
 31.322074615579716 19.737852534147326 25.134649707849572
 19.952195022152488 12.94569843904663   23.618693004455086
 14.695872710062428)
```

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



There's a subtle abstraction in the preceding code that deserves attention. `(require '[edgar.core :as c])` introduces the *quote* symbol. `'` is the reader shorthand for the `quote` function. So, the equivalent invocation would be `(require (quote [edgar.core :as c]))`. Quoting a form tells the Clojure reader not to evaluate the subsequent expression (or `form`). Therefore, evaluating `'(a b c)` returns a list of 3 symbols without trying to evaluate any part of that form. Even though these symbols haven't yet been assigned, it's okay because this expression (or `form`) has not yet been evaluated.

But, this begs a larger question. What is the reader? Clojure (and all Lisps) are what's known as **homoiconic** (you can read more about it at `http://en.wikipedia.org/wiki/Homoiconicity`). This means that Clojure code is also data and data can be directly output and evaluated as code. The reader is what parses our `src/edgar/core.clj` file (or the `(+ 1 1)` input from the REPL prompt) and produces data structures that are evaluated. `read` and `eval` are the two essential processes by which Clojure code runs. The evaluation result is printed (or output) to the standard output device usually. Then, we `loop` the process back to the read function. So, when REPL reads your `src/edgar/two.clj` file, it directly transforms this text representation into data and evaluates it. A few things fall out of this. For example, it becomes trivial for Clojure programs to directly read, transform, and write out other Clojure programs. The implications of this will become clearer when we look at macros. But for now, know that there are ways to modify or delay the evaluation process, in this case by quoting a form.

Basic data structures

Okay! Now we need to assign each price to a point in time in order to produce a time series. Let's think about what this data structure might look like. Assigning the price and time together might look something like `'((t1 29.60706184716407) (t2 12.507593971664075))`. But, this is not quite right. If you recall, evaluating `(t1 29.60706184716407)` would be how we invoke a function. If `t1` is not a function, then your Clojure REPL will produce an error. Also, `t1` and `t2` denote points in time, but Clojure treats them as symbols—which have not been created yet. This will also produce an error. We could delay evaluation by quoting the `'((t1 29.60706184716407) (t2 12.507593971664075)` form`)`. However, we'll still need to create the `t1` and `t2` symbols before we start using this form. There's a better way to do this, as follows:

- Maps are Clojure data structures that associate a key with a value. Evaluating `{0 1}` or `{:a 1}` creates a map that associates the `0` number key with the number value of 1. The second example associates the `:a` keyword with the value of 1 (you can read more at `http://clojure.org/data_structures#Data Structures-Maps`.
- Keywords are symbolic identifiers that evaluate to themselves. So, evaluating `:a`, `:fubar`, and `:thing` will all return themselves. However, trying to evaluate `fubar` or `thing` will produce an error as these represent symbols that have not yet been assigned (for more details on keywords, visit `http://clojure.org/data_structures#Data Structures-Keywords`).
- Symbols are identifiers that are normally used to refer to something else (you can read more about symbols at `http://clojure.org/data_structures#Data Structures-Symbols`). The simplest way to create a symbol is to use the `def` function to bind a value to some name in the current namespace. So, evaluating `(def fubar 1)` or `(def thing 2)` will create these symbols. Now we can successfully evaluate `fubar` or `thing`:
    - `let` is a special form in Clojure that also lets you bind values to symbols within a lexical scope. It's one of the language's fundamental building blocks. You'll see it often in Clojure code as follows:

```
(let [pricelist (c/generate-prices 12 35)]
  (println pricelist))
```

While we can use anything as a key in a map, keywords are the easiest as they provide very fast equality tests. Writing `(def fubar "something")` and then `{fubar 1}`, is more heavy-handed than just writing `{:something 1}`. This leads to another pattern in Clojure of using keywords to assign keys in maps. Therefore, while `{0 29.60706184716407}` is a valid map, `{:time 0 :price 29.60706184716407}` will let us easily manipulate internal data. It is also easier to read and reason about.

Let's play around with this a bit. The following code will give us a lazy infinite sequence of maps with the `{:price <each-price>}` shape:

```
(def pricelist (c/generate-prices 12 35))
(take 25 (map (fn [x] {:price x}) pricelist))
```

Macros and more in-depth data transformation

For each map, we still need to put in `{:time <moment-in-time>}`. Clojure's `map` function has some more useful features as follows:

- It will simultaneously pass a mapping function over an arbitrary number of lists.
- It applies the mapping function to the first item of each collection and iterates until the shortest collection is exhausted. The remaining elements in all other collections are discarded. As such, the mapping function should accept the same number of arguments as collections being mapped over.

These features let us take our lazy infinite sequence of `{:price <each-price>}` and generate a corresponding `{:time <moment-in-time>}` entry. We'll start by using simple integers to denote our moments in time. You can see that the mapping function just returns an `[x y]` vector of each incoming (the `:time` and `:price` entries) argument. If you evaluate the following form, you should see a lazy infinite sequence of vectors in the shape of `([{:time 0} {:price 23.113293577419874}] ...)`:

```
(take 25 (map (fn [x y]
                [x y])
              (map (fn [x] {:time x}) (iterate inc 0))
              (map (fn [x] {:price x}) pricelist)))
```

Vectors are another type of collection where values are indexed by integers (you can read more about vectors at `http://clojure.org/data_structures#Data Structures-Vectors`. Thus, you can efficiently access the (`log32N` hops) elements. Incidentally, vectors are also functions of their elements. So, evaluating `([58 23 4 638] 2)` returns the element at index `2`, which is `4` in this case. Alternatively, you can use the `get` function, `(get [58 23 4 638] 2)`, to also get an element by index.

We currently have two maps per element. Finally, we'll need to transform each `[{:time 0} {:price 23.113293577419874}]` element into something that looks like our original aim of `{:time 0 :price 23.113293577419874}`. This can be accomplished with the `merge` function. So, if you evaluate `(merge {:time 0} {:price 23.113293577419874})`, you'll see the desired structure of `{:price 23.113293577419874, :time 0}` (the order of the entries doesn't matter). We just need to apply this form over each element in our collection of vectors. Therefore, we can again use the `map` function to make the application.

Normally, we would have to introduce an intermediate symbol that binds to `([{:time 0} {:price 23.113293577419874}] ...)`, and then feed this into our second mapping function. However, there's a shortcut that lets us bypass this symbol binding. `->` and `->>` are macros (`http://clojure-doc.org/articles/language/macros.html`) that take an intermediate value and thread it through a series of expressions in either the first or last positions, respectively. This is our first brush with macros. Recall that Clojure, which is homoiconic, can rewrite programs, the results of which are evaluated upon completion. Macros employ this rewriting feature to rewrite code that is given to them. So, `(-> 10 (/ 5))` produces `2` because it places `10` in the `/` function's first argument position, `(/ 10 2)`, while `(->> 10 (/ 5))` produces `1/2` because it places 10 in the `/` function's last argument position `(/ 5 10)`. In this case, `->>` rewrites the following expression to assign the necessary symbols, and then threads the result to the next form. The main point here is that it just makes for more readable code and can be used when intermediate values are only going to be used once. The following expression will give us the structure that we need:

```
(take 25 (->> (map (fn [x y]
                     [x y])
                   (map (fn [x] {:time x}) (iterate inc 0))
                   (map (fn [x] {:price x}) pricelist))
              (map (fn [x] (merge (first x) (second x))))))
```

There are two collections that are important to grasp,`(iterate inc 0)` and `pricelist`, which are first mapped over with mapping functions that create maps using the `:time` and `:price` keys. These two lists are then mapped over using the `(fn [x y] [x y])` mapping function. The result of this is then passed to the second form in the `->>` threading form. The result is then in the form of a list of maps with the `:time` and `:price` entries, for example, `({:time x :price y})`. As such, we can simplify this function to just one `map` call over both collections and return a map with both values. Let's add this expression to a function and make our code a little more manageable:

```
(defn generate-timeseries [pricelist]
  (map (fn [x y]
         {:time x :price y})
       (iterate inc 0)
       pricelist))
```

Your `src/edgar/core.clj` file should look like this:

```
(ns edgar.core)

(defn generate-prices [lower-bound upper-bound]
  (filter (fn [x] (>= x lower-bound))
          (repeatedly (fn [] (rand upper-bound)))))

(defn generate-timeseries [pricelist]
  (map (fn [x y]
         {:time x :price y})
       (iterate inc 0)
       pricelist))
```

Then, in your REPL, you can require the namespace and generate a time series (see the following code). Notice the small size of the code that's required to give us a crude time series. As a rule, object-oriented and procedural languages require many more multiples of code to produce the same result. Since these languages lack FP features, such as laziness, first-class functions, immutability, homoiconicity, and so on, creating this time series would require significantly more code and effort. The fact that code that is this small in size produces results has implications on how fast we can conceptualize and prototype algorithms that model our problem domains:

```
(require '[edgar.core :as c])


(def pricelist (c/generate-prices 12 35))
```

```
  ;;

  (take 10 (c/generate-timeseries pricelist))
de-price {:last 21.417286127291007}} {:last-trade-time #inst
"2015-09-13T20:12:56.560-00:00", :last-trade-price {:last
25.700743352749207}} {:last-trade-time #inst "2015-09-
13T20:12:58.560-00:00", :last-trade-price {:last
20.560594682199365}} {:last-trade-time #inst "2015-\
09-13T20:12:59.560-00:00", :last-trade-price {:last
24.67271361863924}} {:last-trade-time #inst "2015-09-
13T20:12:59.560-00:00", :last-trade-price {:last
19.73817089491139}} {:last-trade-time #inst "2015-09-
13T20:12:59.560-00:00", :last-trade-price {:last
23.68580507389367\
}} {:last-trade-time #inst "2015-09-13T20:13:00.560-00:00", :last-
trade-price {:last 18.948644059114937}} {:last-trade-time #inst
"2015-09-13T20:13:02.560-00:00", :last-trade-price {:last
15.15891524729195}})
```

Elaborating our equation

Now we have a time series of stock prices. If you recall, though, we're choosing random floats within an upper and lower bound, so it in no way represents what we would see in the real world. If I precisely knew the direction of any given stock price, I'd be rich. However, I think we can make some educated approximations on stock price movement.

In the next section, we're going to use `clj-time`, a Clojure date and time library as follows:

1. Add a `[clj-time "0.9.0"]` vector to the `:dependencies` entry of your `edgar/project.clj`.
2. Then, we're going to require some of its namespaces into our code. This can be done at the namespace level. So, change the top of your `src/edgar/core.clj` file to look like this:

   ```
   (ns edgaru.core
       (:require [clj-time.core :as tc]
                 [clj-time.periodic :as tp]
                 [clj-time.coerce :as tco]))
   ```

3. Now, restart your REPL.

A price, of course, is an economic signal of the point at which sellers and buyers are willing to exchange something. What would be useful is to give some crude approximation of the fluctuations that a stock price makes over time. There are a few equations that can produce these kinds of random fluctuations. A stochastic process in probability theory is a collection of random variables that represent the evolution of some system of random values over time (you can read more at `http://en.wikipedia.org/wiki/Stochastic_process`). Here, I'll try to generate prices that stochastically oscillate around its previous levels and a high/low range. This being a test data source, we don't really have input data per se. We do know that our desired data output is a time series, which is similar to what we already have. We just want the price fluctuations to be slightly more realistic. Our first task, then, is to devise an equation that will give us a semi-realistic, random price movement as follows:

1. The first thing to randomize is the direction of the price movement from one tick to the next. This is easy enough as FP allows us to pass around functions like any other variable.
2. The next thing to determine is the size of the price movement. So, instead of any random float between an upper and lower bound, the price needs to move within a *reasonable range* from the last price.
3. Now, we'll take the difference of the last price from its lower bound (if it's below 50% of the high/low range) or from its upper bound (if it's above 50% of the high/low range).
4. The previous price to boundary difference is divided by the overall high/low range, which gives us a percentage of price movement. We'll use this as our next price movement that is randomized by direction.

In terms of code complexity, I've made quite a leap from our preceding simpler function. I just want to show the progression your equations can take. `generate-prices` now has a few helper functions as follows:

- `random-in-range`: This just generates a random initial price within an upper and lower bound. Here, `r` in the `let` binding is a random number between 0 and the input upper bound. If this random number is above our input lower bound, then we'll use it. If not, `(rand (- upper lower))` picks a random floating point number between 0 and the upper and lower bound's difference. Then, we add this (within the difference) random number to the `lower` bound to get the random number within the function's supplied upper and lower bounds.
- `stochastic-k`: This gives us our percentage of price movement of the high/low price.

- `break-local-minima-maxima`: This is a small cheat function that ensures that our *k* values don't go too far above or below 0. Recall that the `->` and `->>` macros put everything in the first or last argument positions of a function. For threaded expressions, we need a way to explicitly place an input value. Clojure's `as->` macro lets us do this by naming and using a Var.
- The `(tc/now)` function call, from the `clj-time` library, gives us the current moment in time.
- `generate-prices` collects some values for calculation, such as `high`, `low`, `k`, `newprice`, and any new high or low values that have occurred:

```
(defn random-in-range [lower upper]
 (let [r (rand upper)]
    (if (>= r lower)
      r
      (+ (rand (- upper lower))
         lower))))


(defn stochastic-k [last-price low-price high-price]
  (let [hlrange (- high-price low-price)
        hlmidpoint (/ hlrange 2)
        numerator (if (> last-price hlmidpoint)
                    (- last-price hlmidpoint)
                    (- hlmidpoint low-price))]
     (/ numerator hlrange)))

(defn break-local-minima-maxima [k]
  (as-> k k
    (if (<= (int (+ 0.95 k)) 0)
      (+ 0.15 k) k)
    (if (>= k 1)
      (- k 0.15) k)))

(defn generate-prices

  ([low high]
   (generate-prices (random-in-range low high)))

  ([last-price]
   (iterate (fn [{:keys [last]}]

              (let [low (- last 5)
                    high (+ last 5)
                    k (stochastic-k last low high)
                    plus-OR-minus (rand-nth [- +])

                    kPM (if (= plus-OR-minus +)
                          (+ 1 (break-local-minima-maxima k))
                          (- 1 (break-local-minima-maxima k)))

                    newprice (* kPM last)
                    newlow (if (< newprice low) newprice low)
                    newhigh (if (> newprice high) newprice high)]

                {:last newprice}))
            {:last last-price}))))


(defn generate-timeseries
  ([pricelist]
   (generate-timeseries pricelist (tc/now)))
  ([pricelist datetime]
   (->> (map (fn [x y] [x y])
          (map (fn [x] {:time x}) (iterate #(tc/plus % (tc/seconds (rand 4))) datetime))
             (map (fn [x] {:price x}) pricelist))
        (map (fn [x] (merge (first x) (second x)))))))
```

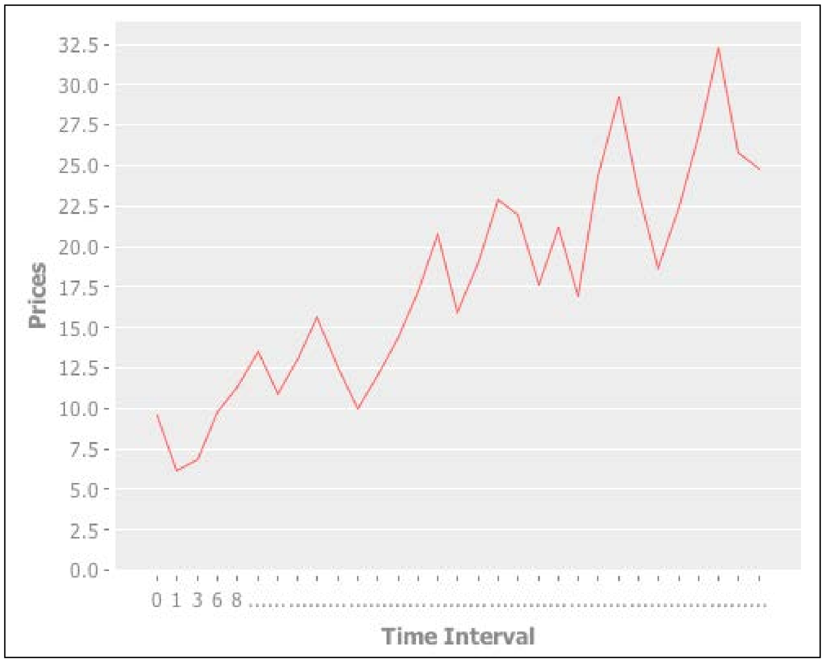
Try evaluating the preceding code and running it with an expression such as `(map :last (take 40 (generate-prices 5 15)))`. You should see something like this:

```
(12.28252717329855 8.944783789632453 12.473307599143377
9.321686446713104 13.350227044440029 11.14774269158978
6.853345365994445 8.123506953556976 10.660889999227342
6.0350125579488845 5.410381179463253 4.376792141566374
4.7619372823332355 5.180974089566862 5.636884932598482
6.6550182288604045 5.813422472457867 4.6667949515838645
5.077459506906932 4.630657649913592 4.2231730733714565
3.766544060739548 3.146182949164578 2.4081361785088022
3.1516592732767386 2.178570011368031 1.459459572689432
2.0318823257034344 1.2349464622890414 1.7416251541648224
2.4561859726269675 1.4484523310344963 2.04272905050674
2.880827959870769 4.06278538620401 5.729680954323048
8.080476992412756 8.783707763945229 7.570532804510094
9.0845037171544)
```

Calling `(def pricelist (generate-prices 5 15))` and then `(take 40 (generate-timeseries pricelist))` should give you a result such as the following . The first record's `:last` price was `15.42`, and the time this price was recorded was on 04/19/2015 at 17:56:58 GMT:

```
  {:price {:last 15.423964294614002}, :time #<DateTime 2015-04-
19T17:56:58.596Z>}
  {:price {:last 15.737626638301053}, :time #<DateTime 2015-04-
19T17:57:00.596Z>}
  {:price {:last 17.94475850354465}, :time #<DateTime 2015-04-
19T17:57:01.596Z>}
  {:price {:last 16.66187358467036)}, :time #<DateTime 2015-04-
19T17:57:02.596Z>}
  {:price {:last 17.521292102081574}, :time #<DateTime 2015-04-
19T17:57:03.596Z>}
  {:price {:last 19.310727826293594}, :time #<DateTime 2015-04-
19T17:57:06.596Z>}
  {:price {:last 16.444691958028105}, :time #<DateTime 2015-04-
19T17:57:09.596Z>}
     )
```

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



Summary

Keep in mind that none of the previous equations are absolutely correct. The point of the exercise is to think about the kind of relationships that each data point has with the other. For a more elaborate equation, feel free to play around with the function, or choose a completely different equation from a stochastically oscillating price. I can see sine waves or polynomials (perhaps, nested) being just as useful. Feel free to play around with the data and share your results.

We've taken pains to first understand what a stream of stock price data may look like. We then applied some core functional programming features that Clojure provides to transform data into the stock price shape we desire. This is an important point to note for any programming language. It should be a tool that empowers you to shape your data as your needs develop. In the next chapter, we're going to continue this lesson, paying close attention to some useful ideas on how to fully capture and represent our data.

3
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.

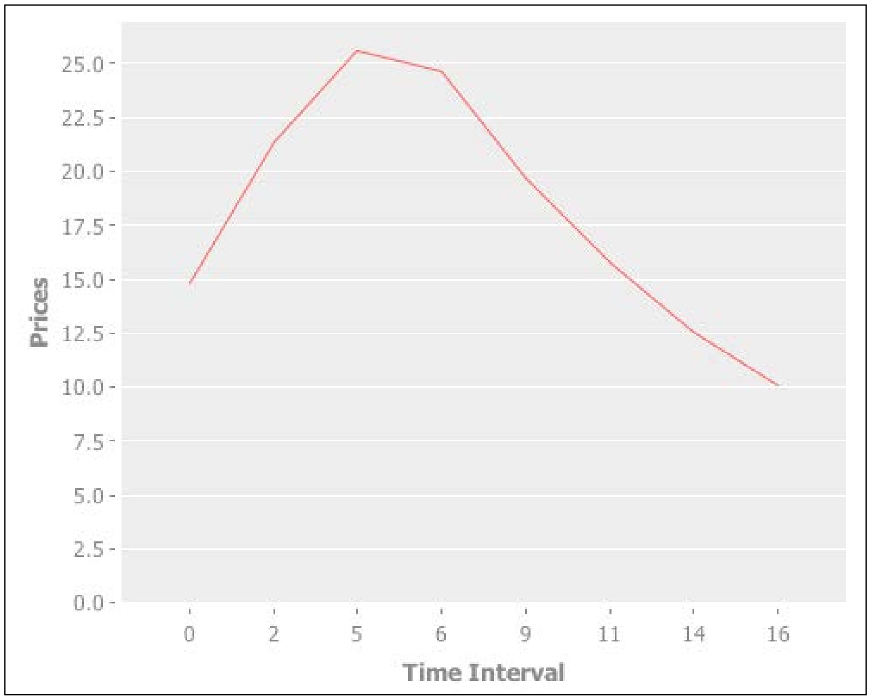
We ultimately need to gauge how close our perception and representations are to actual stock price data. However, for now, we have an infinite stream of price and time points. Apart from this stream of data, we want to calculate a moving average of prices. So, an average is just the sum of a collection 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:

```
({: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-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 10.277830568688747}}
 {:last-trade-time #inst "2015-09-24T04:13:25.868-00:00",
  :last-trade-price {:last 8.222264454950999}}
 {:last-trade-time #inst "2015-09-24T04:13:28.868-00:00",
  :last-trade-price {:last 9.866717345941199}}
...
)
```

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



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:

```
({:average <…>,
  :last-trade-time #inst "2015-09-24T04:13:13.868-00:00",
  :last-trade-price {:last 5.466160487301605}}
 {:average <…>,
  :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 *(/ price-sum (count ticks-within-window))*.

Here, `price-sum`, as the name implies, is the sum of all the prices with our chosen average range, and `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}
        ...)
```

```
            (  {: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`):

```
(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, `(take 40 (generate-prices 5 15))` or `(take 2 our-average)`, were the expressions actually run. This means that `(def timeseries (generate-timeseries 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`. 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`) 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:

```
(reduce (fn [accumulated-result-so-far each-item-list]
          ;; ** single line comments come after semi-colons
```

```
            ;; 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:

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

Now we can start to write our function as follows:

```
(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]
              (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})]))))
            '()
            (partition tick-window
                      1
                      (take (* 2 tick-window) tick-list)))))
```

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>}
  ```

```
    {:last-trade-price {:last 17.94475850354465},
    :last-trade-time #<DateTime 2015-04-19T17:57:01.596Z>}
    {:last-trade-price {:last 16.66187358467036},
    :last-trade-time #<DateTime 2015-04-19T17:57:02.596Z>}
    {:last-trade-price {:last 17.521292102081574},
    :last-trade-time #<DateTime 2015-04-19T17:57:03.596Z>}
    {:last-trade-price {:last 19.310727826293594},
    :last-trade-time #<DateTime 2015-04-19T17:57:06.596Z>}
    {:last-trade-price {:last 16.444691958028105},
    :last-trade-time #<DateTime 2015-04-19T17:57:09.596Z>}
    ...)
```

- `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 5.466160487301605}#inst "2015-09-24T04:13:13.868-
00:00")

(zipmap [:a :b] [1 2])
;; {:b 2, :a 1}

(zipmap etal-keys
        (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:

```
(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:

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

(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:

```
(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 `concat` function joins the contents of two lists and puts the addition on the right-hand side. The `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 `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:

```
(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}))
;; (1
;;  [:last-trade-price-exponential 6.0972436]
;;  [:last-trade-time #inst "2015-06-28T18:31:51.810-00:00"]
;;  [:last-trade-price 6.111604269033163])

(seq {:a 1 :b 2})
;; ([:b 2] [:a 1])
```

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.

```
;; 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
;;  {:last-trade-price-average 6.0972436,
;;   :last-trade-time #inst "2015-06-28T18:31:51.810-00:00",
;;   :last-trade-price 6.111604269033163})
```

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
```

```
            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]
       ...
   )
```

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` 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:

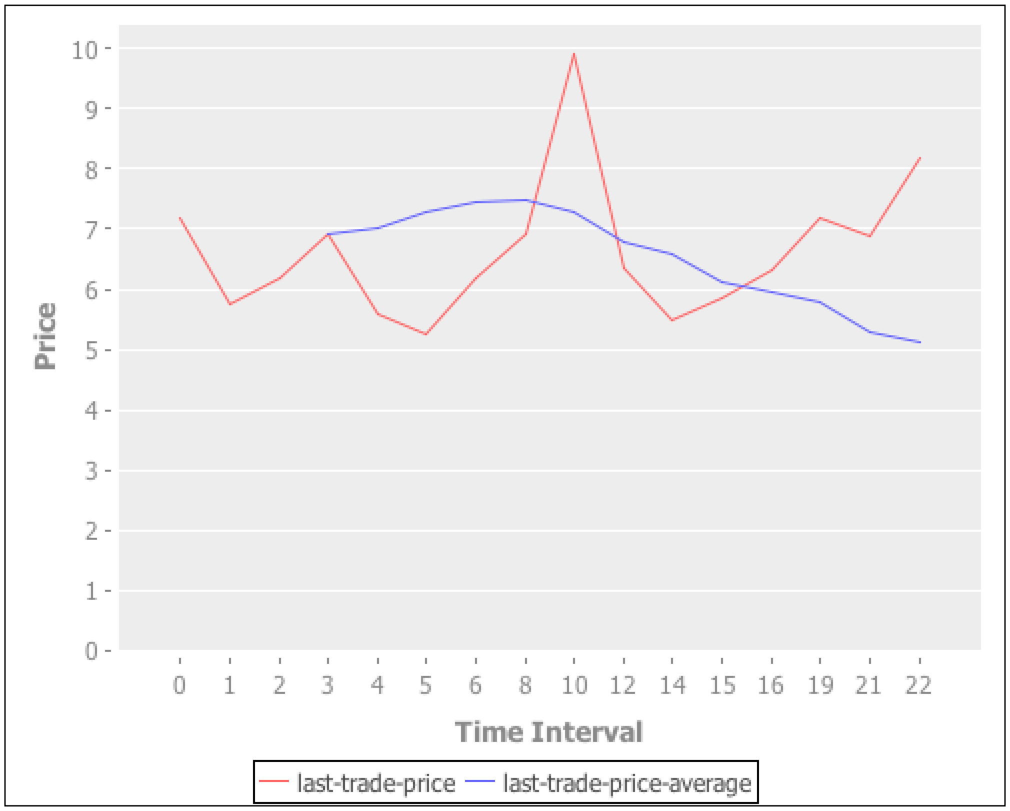
```
(let [{input-key :input
       output-key :output
       etal-keys :etal
       :or {input-key :last-trade-price
            output-key :last-trade-price-exponential
            etal-keys [:last-trade-price :last-trade-time]}}
      {:input :fubar
       :output :thing
       :etal [:one :two]}]

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

;; :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:

```
(let [{input-key :input
       output-key :output
```

```
          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):



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.

4
Strategies for Calculating and Manipulating Data

The previous chapter gave us our first introduction to building an equation: a simple moving average. In this chapter, we're going to implement two more functions where the math is only slightly more advanced than the **simple moving average** (**SMA**). The **exponential moving average** (**EMA**) and **Bollinger Bands** are both technical trading indicators. These, coupled with the SMA, will give us a nice set of lagging indicators on top of our base stream of stock price data. The moving averages lag price movement because they're based on past data. They smoothen price data and show its current direction by filtering out noise or erratic price points in a time series.

Our first refactor – the price list

Our exponential moving average calculation is calculated on every tick increment over our time series. Again, we'll employ the `reduce` function to fold up the previous 20 ticks, into an average value. In order to do this, though, we'll need to dig into the source data for the SMA.

If you recall from *Chapter 2*, *First Principles and a Useful Way to Think*, our prices were a list of maps that looked like the following piece of code. We put the price in a map to communicate the fact that it was the last or most recent price being calculated:

```
{:last 16.68101235523256}
```

Our final time series was a list of maps that looked like the following:

```
{:last-trade-price {:last 5.466160487301605},
 :last-trade-time #inst "2015-09-24T04:13:13.868-00:00"}
```

This means that our price lists aren't just carrying around prices. They're carrying around price and time values. Now we'll add our calculated average to the stuff carried around. You'll often find the need to carry around essential data as well as some extra stuff (also known as a local or dangling state) that's needed to perform future calculations, regression testing, and so on.

In math, to factor a number refers to breaking it up into other numbers that can be operated on together to get the original number. This is a good way to explain how code is factored. Code that is well factored is composed of several pieces that, when changed or modified, does not affect other pieces of code. It is, therefore, easy to compose small pieces of code into a larger whole. This is referred to as **orthogonality** (you can read more about it at `https://en.wikipedia.org/wiki/Orthogonality_(programming)`) and usually means writing code that does only one thing very well. The result of that piece is passed on to the next piece, which does its job, and so on. Discerning the amount of stuff to carry around informs us on how to design our algorithms and vice versa. Ideally, we should only carry around the necessary amount of stuff needed for a given calculation or set of calculations. The `:last` key and accompanying map are artifacts of the oscillating stochastic function that we used to generate each price. In this case, they are no longer necessary when calculating the SMA.

Let's make a function to strip the `:last` key and accompanying map, out of our time series. Then we'll slightly adjust the SMA function so that you do not have to deal with the unnecessary structure. Remember that our original price list looks like this:

```
({:last 10.978625695681702}
 {:last 15.393542022616002}
 {:last 15.68497326182313}
 {:last 17.894866781714637}
 {:last 19.178454686228328}
 ...)
```

Our keywords also act as functions. So, all we really need to do is traverse or `map` over the original price list (a lazy sequence) and extract only the `:last` (price) entry. Here's how we do this. Put this `extract-price-only` function into your `analytics.clj` file:

```
(defn extract-price-only [pricelist]
  (map :last pricelist))
```

In our `edgar.analytics` namespace, as we call it now, we can pass in results from the `generate-prices` function in the `core` namespace. Here's how our `price-only` list will look:

```
(def price-only-list (extract-price-only (core/generate-prices 5 15)))
```

```
'(10.978625695681702
  15.393542022616002
  15.68497326182313
  17.894866781714637
  19.178454686228328
  ...)
```

We can even put that into a convenience function:

```
(defn generate-prices-essential [beginning-low beginning-high]
  (extract-price-only (core/generate-prices beginning-low beginning-high)))
```

Now our time series looks a little cleaner. We can also use a cleaner input list in our original SMA function as follows:

```
(def time-series (core/generate-timeseries price-only-list))

'({:last-trade-price 10.774174002394385,
   :last-trade-time #inst "2015-06-27T17:54:37.583Z"}
  {:last-trade-price 6.221195542189912,
   :last-trade-time #inst "2015-06-27T17:54:40.583Z"}
  {:last-trade-price 6.98092516851132,
   :last-trade-time #inst "2015-06-27T17:54:40.583Z"}
  {:last-trade-price 5.5980561319315,
   :last-trade-time #inst "2015-06-27T17:54:40.583Z"}
  {:last-trade-price 5.263260952271663,
   :last-trade-time #inst "2015-06-27T17:54:42.583Z"})
```

The exponential moving average

An **exponential moving average** (**EMA**) basically weights newer prices more than older ones, thereby reducing lag and applying more weight to recent prices (for more details on EMA, visit `https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average`). This way, more recent price changes have a greater effect on the overall average. Let's build our EMA function. To begin, I'll use raw algebra to outline the equation:

1. The EMA is a function of the SMA. Thus, we use the SMA as the price input to our function.
2. We also have to decide the time period (or `tick-window`) for which our EMA will apply. Since our SMA has used 20 time periods (or ticks), we're also going to use this number for our EMA.

3. Next, we'll calculate our constant. This constant is typically denoted as *k*. This is then 2 divided by our time period plus 1. Mathematically, the equation is *k = 2 / (tick-window + 1)*. So, a time period of 20 gives us *2 / (20 + 1)*, which equals 0.04761904761905.
4. Finally, the EMA denotes the price, which is now multiplied by *k* plus the previous EMA multiplied by 1 minus *k*. So, let's say that the most recent nominal prices are `'(9.8 10)` and the SMA is (`9.6 9.7`):

   This would make the EMA *(10 * 0.04761904761905) + (9.6 * (1 - 0.04761904761905))*.

   This equates to *0.47619047619 + 9.14285714286*, which then equals 9.61904761905.

Working backwards from a point on our *EMA = ((price(now) * k) + (ema(previous) * (1 - k)))*, we see that we need these pieces of data:

- **price(now)**: This is in our original time series, which is now a part of the SMA list.
- **k**: This is easily calculated using simple algebra, *k = 2 / (tick-window + 1)*.
- **ema(previous)**: One of the first calculations, we'll use the previous SMA (`9.8` in this case). Thereafter, we'll calculate it (using the preceding equation) and pass on the result to our next calculation.

We're going to use two `let` blocks because one is needed inside reduce's accumulator function. However, we don't need to recalculate *k*, `input-key`, and so on for each item traversal. So, we put that into a more globally scoped `let` block. With these bits broken out, we should be able to step back and look at the entire EMA function:

```
(defn exponential-moving-average
  "From a tick-list, generates an accompanying exponential moving average list.
     EMA = price(today) * k + EMA(yesterday) * (1 - k)
     k = 2 / N + 1
     N = number of days
   Returns a list, equal in length to the tick-list, but only with slots filled,
   where preceding tick-list allows.
   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-exponential)
   :etal - other keys to emit in each result map
```

```
 ** This function assumes the latest tick is on the left**"

([options tick-window tick-list]
(exponential-moving-average options tick-window tick-list (simple-moving-average {} tick-window tick-list)))

([options tick-window tick-list sma-list]

 ;; 1. calculate 'k'
 ;; k = 2 / N + 1
 ;; N = number of days
 (let [k (/ 2 (+ tick-window 1))

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

   ;; 2. get the simple-moving-average for a given tick - 1
   (reduce (fn [rslt ech]

         ;; 3. calculate the EMA ( for the first tick, EMA(yesterday) = MA(yesterday) )
             (let [;; price(today)
                   ltprice (input-key ech)

                   ;; EMA(yesterday)
                   ema-last (if (output-key (first rslt))
                                (output-key (first rslt))
                                (input-key ech))

                ;; ** EMA now = price(today) * k + EMA(yesterday) * (1 - k)
                   ema-now (+ (* k ltprice)
                              (* ema-last (- 1 k)))]

                (lazy-cat rslt
                          [(merge
                            (zipmap etal-keys
                            (map #(% (last (:population ech))) etal-keys))
                            {output-key ema-now})])))
           '()
           sma-list))))
```

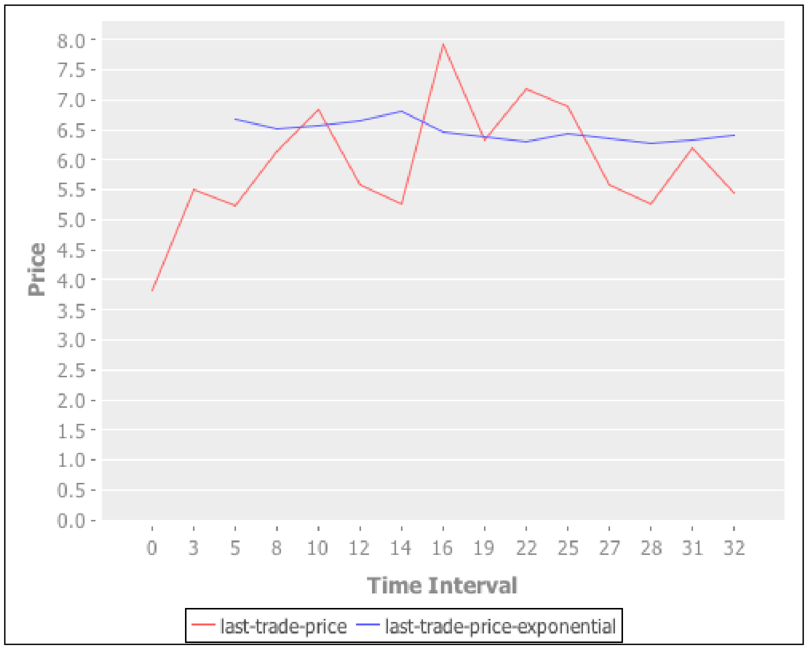
Now we can call this function with a timeseries (created from a pricelist):

```
(def price-only-list (extract-price-only (core/generate-prices 5 15)))
(def time-series (core/generate-timeseries price-only-list))

(def ema-list (exponential-moving-average {} 20 (take 40 time-series)))
```

You should see a result that looks something like this:

```
  ({:last-trade-price 6.994839831563898,
   :last-trade-time #inst "2015-09-25T00:56:57.133-00:00",
   :last-trade-price-exponential 9.861794696331243}
  {:last-trade-price 8.390198342705236,
   :last-trade-time #inst "2015-09-25T00:57:00.133-00:00",
   :last-trade-price-exponential 9.673950987829697}
  {:last-trade-price 6.71215867416419,
   :last-trade-time #inst "2015-09-25T00:57:03.133-00:00",
   :last-trade-price-exponential 9.523676021028459}
  {:last-trade-price 7.861386743797853,
   :last-trade-time #inst "2015-09-25T00:57:06.133-00:00",
   :last-trade-price-exponential 9.602513190297485}
...)
```

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



The Bollinger Bands

The Bollinger Band is another technical trading indicator. It provides a volatility range for a stock price. This is based on the standard deviation from a price's mean. The standard deviation changes as volatility increases and decreases. This translates to a widening of the band when volatility increases and a narrowing of the band when volatility decreases. You can read more about Bollinger Bands at `https://en.wikipedia.org/wiki/Bollinger_Bands` and `http://www.investopedia.com/terms/b/bollingerbands.asp`.

The Bollinger Band has upper, lower, and middle bands. For us, the middle band will be the SMA. The upper and lower bands will be two standard deviations from the mean. The standard deviation is the square root of a variance. The variance is the average of the squared differences from the mean. Let's take a closer look at the math. Again, we'll first note down the equation using an algebraic pseudocode:

1. We'll start with a list of SMAs and our middle band.
2. Then, we'll calculate the average (or the mean) of the last 20 SMAs.
3. For each mean, we calculate its difference from the nominal price and square this difference like this *(mean - last-trade-price)*^2.
4. This time, we'll calculate the average of the squared differences to get the variance.
5. Then, simply take the square root of this variance to get the standard deviation.

We're going to similarly work backwards from the data points that we need:

- **Middle Bollinger Band**: This refers to our SMA list.
- **Variance**: This refers to *(sum-of-the-price-mean-difference-squared % number-of-means)*. Here, the % is the remainder operator.
- **Standard deviation**: This refers to the square root of the variance.
- **Upper Bollinger Band**: This refers to *(moving-average* + *(2 * standard-deviation))*.
- **Lower Bollinger Band**: This refers to *(moving-average - (2 * standard-deviation))*.

From here, we can take a similar approach of reducing the SMA and calculating the variables that we need. I've included the following completed function as a guide for you to follow. However, see if you can build it on your own, patterning your solution after the first two functions we've completed:

```
(defn bollinger-band
  "From a tick-list, generates an accompanying list with upper-band and lower-band
```

```
Upper Band: K times an N-period standard deviation above the moving average (MA + Kσ)
Lower Band: K times an N-period standard deviation below the moving average (MA − Kσ)
K: number of standard deviations
N: period, or tick-window we are looking at
Returns a list, equal in length to the tick-list, but only with slots filled,
where preceding tick-list allows.
** This function assumes the latest tick is on the left**"

([tick-window tick-list]
(bollinger-band tick-window tick-list (three/simple-moving-average nil tick-window tick-list)))

([tick-window tick-list sma-list]

;; At each step, the Standard Deviation will be: the square root of the variance (average of the squared differences from the Mean)
 (reduce (fn [rslt ech]

           (let [;; get the Moving Average
                 ma (:last-trade-price-average ech)

                 ;; work out the mean
                 mean (/ (reduce (fn [rslt ech]
                                   (+ (:last-trade-price ech)
                                      rslt))
                                 0
                                 (:population ech))
                         (count (:population ech)))

;; Then for each number: subtract the mean and square the result (the squared difference)
                 sq-diff-list (map (fn [ech]
                              (let [diff (- mean (:last-trade-price ech))]
                                   (* diff diff)))
                                   (:population ech))

              variance (/ (reduce + sq-diff-list) (count (:population ech)))
                 standard-deviation (. Math sqrt variance)]
```

```
                  (lazy-cat rslt
                            [{:last-trade-price (:last-trade-price ech)
                              :last-trade-time (:last-trade-time ech)
                              :upper-band (+ ma (* 2 standard-deviation))
                           :lower-band (- ma (* 2 standard-deviation))}]))))
              '()
              sma-list)))
```

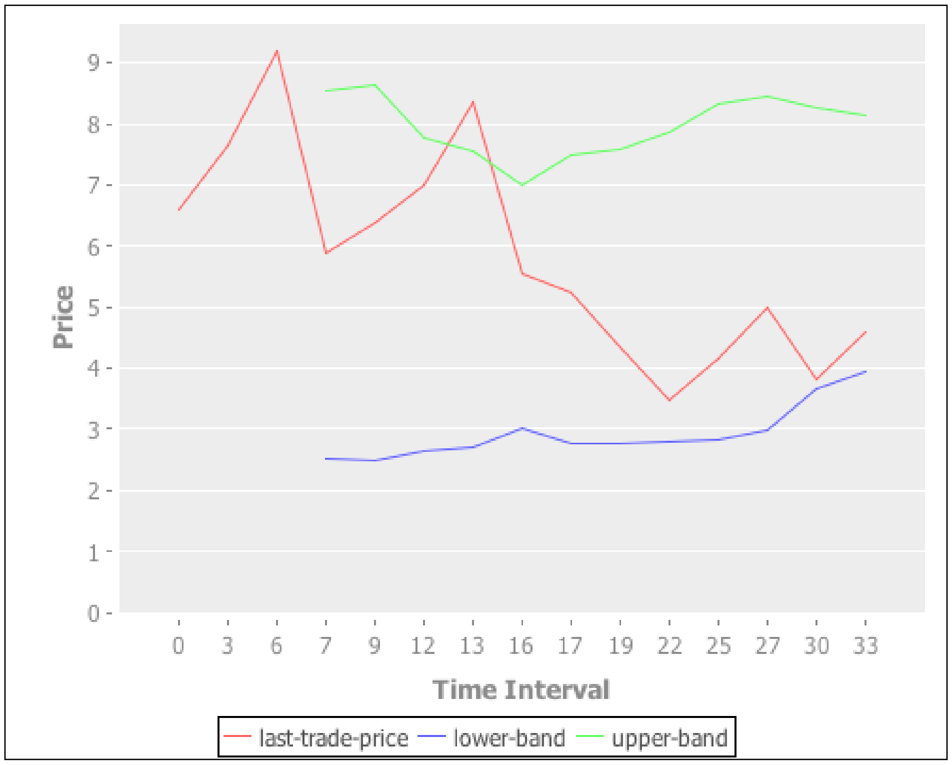
The `timeseries` collection should already be defined, so now we can call our `bollinger-band` function in a similar way to the `exponential-moving-average` function:

```
(def bol-band (bollinger-band 20 (take 40 time-series)))
```

You should see a result that looks something like this:

```
({:last-trade-price 14.72248400774257,
  :last-trade-time #inst "2015-09-25T00:59:16.924-00:00",
  :upper-band 24.882020638795275,
  :lower-band 9.546205067243378}
 {:last-trade-price 17.666980809291083,
  :last-trade-time #inst "2015-09-25T00:59:19.924-00:00",
  :upper-band 24.882647571351672,
  :lower-band 9.541527911594255}
 {:last-trade-price 21.2003769711493,
  :last-trade-time #inst "2015-09-25T00:59:21.924-00:00",
  :upper-band 24.87967020977339,
  :lower-band 9.53964500546127}
 ...)
```

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



Summary

In this chapter, I've intentionally spent most of my time exploring the math in each equation and the algorithmic steps needed to arrive at a desired endpoint. By now, you can see how Clojure is a tool that lets you easily represent data and quickly perform calculations. As with Christophe Grande's suggestion, we want to let data drive the design of our software. We also want a loose coupling of our software, which will allow an orthogonal design. This means that small pieces will be easily composed into a larger whole. This also means minimizing the amount of baggage data or extra stuff (also known as local or dangling state) we carry in our data structures to only allow what's necessary in performing a given calculation.

In *Chapter 5*, *Traversing Data, Branching, and Conditional Dispatch*, we're going to do a larger code refactoring of our original time series generator. Let's see what kind of cleanup we can perform there.

5 Traversing Data, Branching, and Conditional Dispatch

In this chapter, we'll be looking at more advanced branching and conditional logic. However, these are just the implementation details of the kind of control we're beginning to realize. Alan Kay, a notable computer programmer, has stated that, "Lisp isn't a language, it's a building material." The fact that Clojure is a Lisp fits this description. It also helps us understand that Clojure should just be the starting point when building functions and engineering our systems. It is the kind of building material that lets us be as precise as we need.

Our second refactor – the generate prices function

For example, our current oscillating stochastic price generator isn't the best fit for our needs. The generated prices either regularly trend to zero or trend much too high:

```
(defn random-in-range [lower upper]

  (let [r (+ (rand (- upper lower))
             lower)]

    (if (> r upper)
      upper r)))

(defn stochastic-k [last-price low-price high-price]
  (let[hlrange (- high-price low-price)
       hlmidpoint (/ hlrange 2)
       numerator (if (> last-price hlmidpoint)
                   (- last-price hlmidpoint)
                   (- hlmidpoint low-price))]
    (/ numerator hlrange)))

(defn break-local-minima-maxima [k]
  (as-> k k
    (if (<= (int (+ 0.95 k)) 0)
      (+ 0.15 k) k)
    (if (>= k 1)
      (- k 0.15) k)))

(defn generate-prices

  ([low high]
   (generate-prices (random-in-range low high) low high))

  ([last-price low high]
   (iterate (fn [{:keys [last lows highs]}]

              (let [low (-> lows first)
                    high (-> highs reverse first)
                    k (stochastic-k last low high)
                    plus-OR-minus (rand-nth [- +])

                    kPM (if (< k 0.5)
                          (if (= plus-OR-minus +)
                            (+ 1 (break-local-minima-maxima k))
                            (- 1 (break-local-minima-maxima k)))
                          (if (= plus-OR-minus +)
                            (+ 1 (- 1 (break-local-minima-maxima k)))
                           (- 1 (- 1 (break-local-minima-maxima k)))))

                    newprice (* kPM last)
                    newlow (if (< newprice low) newprice low)
                    newhigh (if (> newprice high) newprice high)]

         (println (str "[" last " | " low " | " high "] <=> k[" k "] / kPM[" kPM "] / newprice[" newprice "]"))
                {:last newprice
                 :lows (take 5 (conj lows newlow))
                 :highs (take 5 (conj highs newhigh))}))

            {:last last-price :lows [low] :highs [high]})))
```

The preceding code shows why we need the `break-local-minima-maxima` guard function: to keep our prices within a reasonable range. However, this is a less elegant workaround, akin to a set of guardrails. You can evaluate the function in your REPL using the following code:

```
(require '[edgar.core :as core]

         '[edgar.analytics :as analytics]


(def price-only-list
  (analytics/extract-price-only (core/generate-prices 5 15)))


(take 50 price-only-list)

;; (6.223646585866444 6.985200975509927 8.371903754581446
11.194829124888214
6.935005351100434 5.593078104671486 5.261364888511671
6.188083226538929
5.45288743795115 4.3879999001375065 4.00841221377164 3.465993865670304
2.7745356087257838 3.5321301673360033 2.567672336389048
1.814142431466118
2.471619508589675 3.367377825071922 4.587774686750374 6.25046480370463
8.51574301919856 7.273943476527238 7.649623431542617 8.26265266318511
9.308951577246637 11.22640931794612 7.281474468812783
6.901246804698159
5.7046915345434 7.772171932425898 8.467256074658573 9.670849322422702
11.928275932423938 7.101779616374013 8.441108859931914
9.624246864446237
7.411713217948748 6.951478047181248 5.719732052754735
7.792663398578526
7.083636475728704 8.409797330774097 7.251018224192089
5.801476360694858
3.698919442080443 2.3583660758658964 1.5036527923589607
0.9193511112141406
0.5197016575549582 0.7609002592527179)
```

We may want to keep this function around as stock prices do sometimes flatline to zero or skyrocket as well. However, we also need a function that more directly keeps prices within a reasonable range for long periods. We should be able to choose or combine both functionalities now that we know we can get more finely grained control over our input data. The only barrier now is figuring out which function can give us a more consistent oscillation. Are there any math functions that give us the kind of wave formation that we usually see in stock graphs? For this, both polynomials and sine waves approximate the oscillating behavior that we're interested in. Also, having a powerful building material at our disposal, we can spend our time investigating their mathematical properties, confident that Clojure will be able to clearly and directly implement any solution we derive.

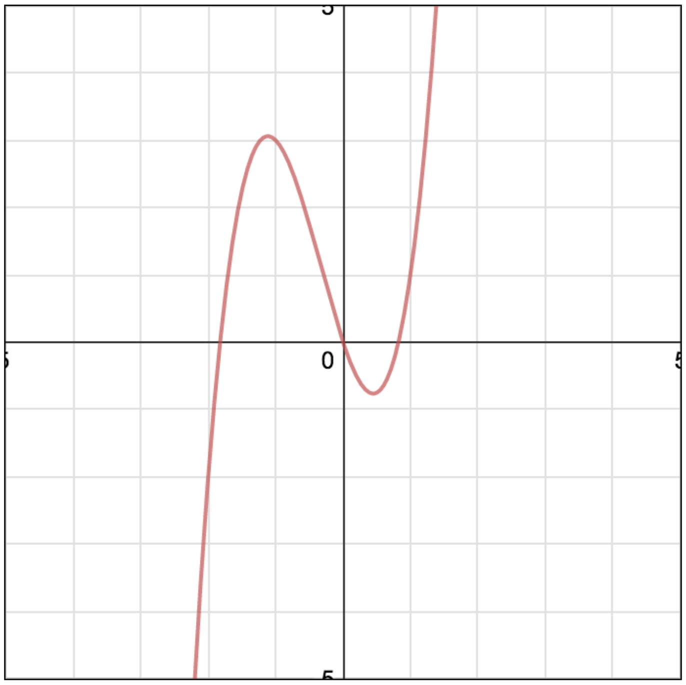
In both cases, these are several desirable properties that we'll want our function to have:

- Can the function's graph repeat, to get continuous waves?
- Can we make the waves slimmer or wider?
- Can we make the waves taller or shorter?
- Are we able to nest waves within waves?

Polynomial expressions

A polynomial is an expression of more than two algebraic terms. These terms have variables and coefficients that only involve addition, subtraction, and multiplication. The variables can only have non-negative exponents, and these exponents typically decrease in size from the left-hand side to the right-hand side of the equation. So, for example, *(2 * x^3) + (2 * x^2) - (3 * x)* is a typical polynomial equation.

Polynomial expressions that have the right variables and coefficients are interesting for our purpose because they graph as a nice upward and downward wave:



If we try to encode the preceding polynomial expression, we'll get something like the following `polynomial` function in this section. We've seen the threading macro before, which is `->`. This takes some data (or an expression that outputs some data) and places it in the first parameter position of the next expression. If we used `->>`, it would place the data in the last parameter position of the next expression. The algebra also translates pretty directly from a source equation. Take the power of a variable, multiply it by the coefficient, then add the result with the next term. Take this result and subtract the subsequent term from it. Open your editor and create a file called `datasource.clj` in the `src/edgar` directory. We're going to pull in and use some library functions. To do this, create a namespace form as follows:

```
(ns edgar.datasource
  (:require [edgar.core :as core]
                [edgar.analytics :as analytics]
                [clojure.math.numeric-tower :as math])
  (:import [org.apache.commons.math3.distribution BetaDistribution]))
```

In order to use the `clojure.math.numeric-tower` Clojure namespace and the `org.apache.commons.math3.distribution` Java package, we have to include these libraries in our `project.clj` file. It should look like the following piece of code. After you're done editing `project.clj`, restart any REPLs you may have opened:

```
(defproject edgaru "0.1.0-SNAPSHOT"
  :description ""
  :url "https://github.com/twashing/edgaru"
  :license {:name "Eclipse Public License"
                :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.7.0"]
                          [clj-time "0.9.0"]
                        [org.clojure/math.numeric-tower "0.0.4"]
                        [org.apache.commons/commons-math3 "3.5"]])
```

Our next function, `polynomial`, goes into `src/edgar/datasource.clj` and looks like this:

```
(defn polynomial [x]
  (->
   (+ (* 2
         (Math/pow x 3))
      (* 2
         (Math/pow x 2)))
   (- (* 3 x))))
```

If we try to map this function over a range from `-5` to `5`, we'll get the following result. If we compare the result with our graph, we'll see that the values match. When *x* is `-5`, *y* will be `-185`, which is way off the graph. We have to get to *x* of `-2` to see *y* of `-2` become visible in our graph. When *x* of `1`, puts *y* at `3`, which matches the left hump we see (let's call this the local maxima) before the graph curves back down again to *x* of `0` and *y* of `0`. When *x* is `1`, *y* is `1`, and when *x* is `2`, *y* swings up and out of our view again to `18`:

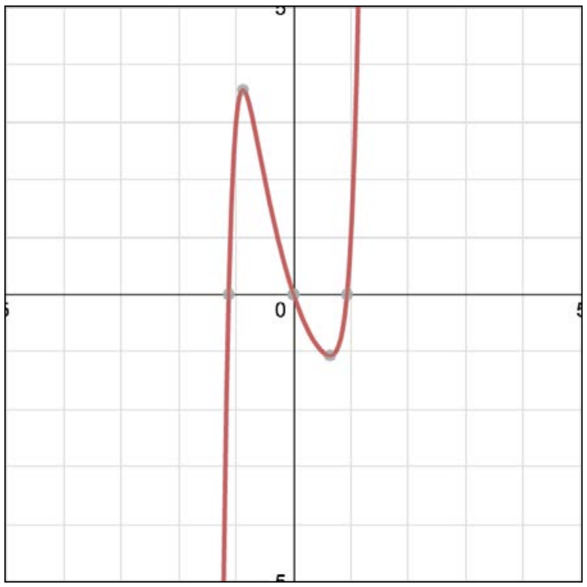
```
=> (map polynomial '(-5 -4 -3 -2 -1 0 1 2 3 4 5))
(-185.0 -84.0 -27.0 -2.0 3.0 0.0 1.0 18.0 63.0 148.0 285.0
```

The preceding code shows the sharp swings we see, and we're going to have to get more granular in order to better capture some of y's changes. Let's zoom in and get the *y* values at each tenth increment of *x* from `0` to `1`. We can see *y* decrease (to a local minima) from *x* of `0` to `0.5`. This decrease also shrinks from `0.3` to `0.5` (-0.6659 -0.752 -0.75). After this, `0.6`, `0.7`, and `0.8` start to swing *y* up again:

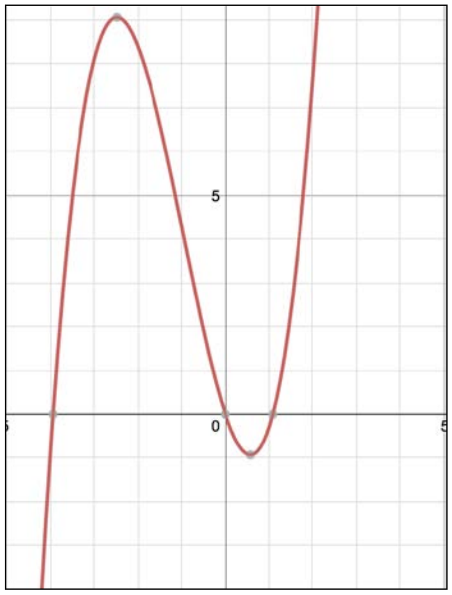
```
=> (map polynomial '(0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1))
(0.0 -0.278 -0.5040000000000001 -0.6659999999999999
-0.7520000000000001 -0.75 -
0.6479999999999999 -0.43399999999999994 -0.09600000000000009
0.3780000000000001 1.0)
```

So far, we have been able to easily create a polynomial function that could give us some interesting price swings. Let's see how it fits our needs:

- **Repeatable**: We can repeat the swings in the x's right before and after the local y maxima and minima.
- **Slimmer or wider waves**: If we give the first exponent a larger odd value, the hump on the left of our curve gets slimmer. However, we then start to lose any symmetry that we have on the left versus the right-hand side. This is not necessarily a bad thing as price movements are seldom symmetrical. But, at the very least, we'd like to be able to control this as much as possible:



- **Taller or shorter waves**: I've hard-coded the coefficients and exponents used in this expression. However, when playing with the numbers, notice that smaller coefficients in the first term (from 2 to 0.4) make the overall wave grow (by mostly making it taller and a little wider). Similarly, larger coefficients in the next two terms also make the wave taller and slightly wider. The opposite happens when we perform the associated opposite actions on each term:



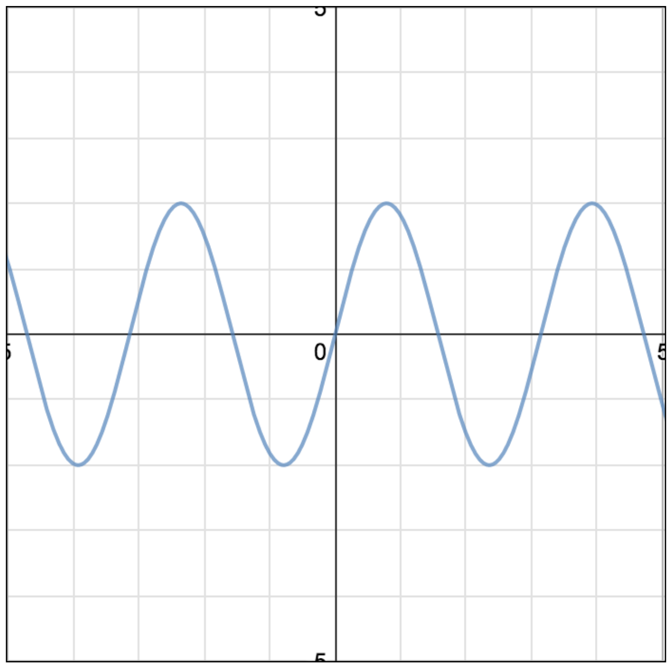
- **Nestable waves**: Since the *y* values swing so sharply, I think it would be difficult to place a polynomial wave within a wave at, say, between *x* values of -1 and 0. This doesn't improve much if we make the wave taller (and a little wider).

So, this function has some interesting properties. Strikingly, though, our `polynomial` function, even though we've just started, is much smaller and more direct (as Clojurians often say, "easier to reason about") than the original `generate-prices` function. This is a good sign. Again, we've spent most of our time thinking about the problem (the math, in this case). Clojure was just the medium that let us quickly translate our equation.

A sine wave

A sine is a trigonometric function of an angle of a right triangle. The sine (of an angle) is the ratio of the opposite side's length to the length of the hypotenuse (the longest side of the triangle).

A sine wave is a repetitive oscillating graph of the sine function. Therefore, as the function of the angle changes (usually by a clockwise rotation of the hypotenuse), the resulting *y* value follows. It provides a natural upward and downward wave that we're looking for:



It's specific equation is as follows. Let's take a closer look at it and its derivation:

$f(x) = a \sin (b(x - c)) + d$

$y = a \sin (b(x - c)) + d$

$y = a \sin (b(x - Pi/2)) + d$

The variables in the equation are described here:

- **f(x)**: The value of this is *y*
- **a**: The amplitude of the wave (this makes the wave taller or shorter)
- **b**: The horizontal dilation (this makes the wave wider or thinner)
- **c**: *Pi/2* is the horizontal translation (this moves the wave to the left or right-hand side)
- **d**: The vertical translation (this moves the wave in an upward or downward direction)

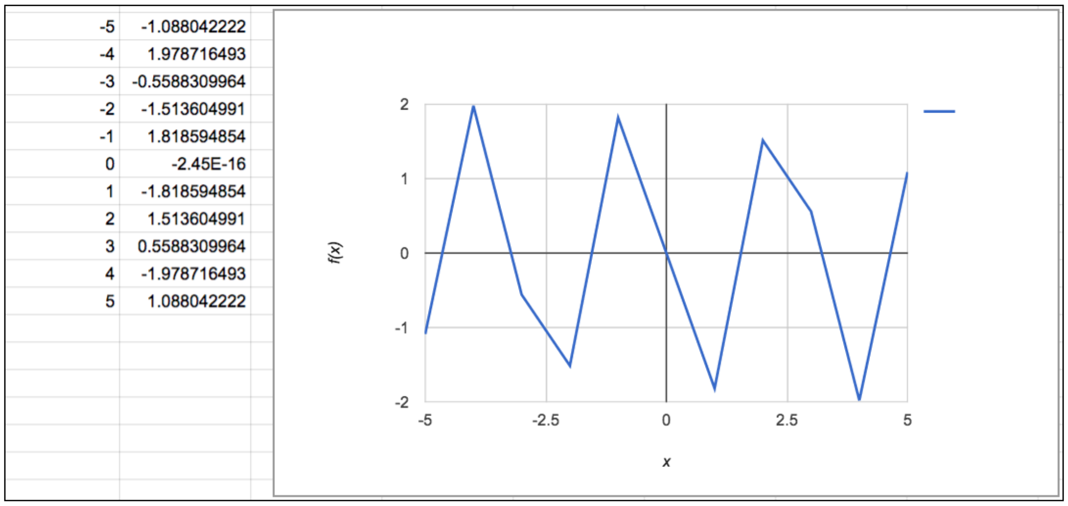
Once we've derived and understood the equation $y = a \sin (b(x - Pi/2)) + d$, we can translate it again pretty quickly. The following code is a simple Clojure implementation, which is based on the algebra we've written. Note that `Math/PI` is a constant value that approximates Pi. The `Math` class is an alias to the `java.lang.Math` Java class, in which the `PI` variable is a static (or class-level) member. This is an example of Clojure's interoperability with the underlying Java platform:

```
(defn sine [a b x d]
  (- (* a
        (Math/sin (* b
                     (- x
                        (/ Math/PI 2)))))
     d))
```

Mapping this function over an integer range of `-5` to `5` provides *y* values that look approximately like what we saw in the previous graph:

```
(map #(sine 2 2 % 0) '(-5 -4 -3 -2 -1 0 1 2 3 4 5))
(-1.0880422217787393 1.9787164932467636 -0.5588309963978519
-1.5136049906158564
1.8185948536513634 -2.4492935982947064E-16 -1.8185948536513632
1.5136049906158566
0.5588309963978515 -1.9787164932467636 1.0880422217787398)
```

If we graph our numbers, the graph looks a bit choppier. However, this is because we've only plugged in whole integer values for *x*:



-5	-1.088042222
-4	1.978716493
-3	-0.5588309964
-2	-1.513604991
-1	1.818594854
0	-2.45E-16
1	-1.818594854
2	1.513604991
3	0.5588309964
4	-1.978716493
5	1.088042222

Getting more granular with our *x* values will again smooth out our curve. This will matter as we determine what our price values should look like. So, let's once again see how the sine function fits our needs:

- **Repeatable**: By the very nature of the clockwise rotation of the sine phases, the graph of the function repeats itself.
- **Slimmer or wider waves**: This is the horizontal dilation or *b* in our equation.
- **Taller or shorter waves**: This is the vertical dilation or amplitude *a* in our equation.
- **Nestable waves**: This will again be difficult because y's slope is often quite steep. However, because we can repeat the wave and dilate it vertically and horizontally, this property might not be as important to generate interesting price movements.

As well as we fared with the polynomial function, our sine function seems to give us even better control facilities. It is easier to separately vertically and/or horizontally dilate the graph. The sine Clojure function is also, again, much smaller and more direct than the original `generate-prices` function.

Stitching the pieces together

While the sine function has so far given us the most amount of control, there are properties of all those functions that would be nice to incorporate into our stream of price data. What we will aim for is to randomly include data samples from the sine, polynomial, and oscillating stochastic functions.

To do this, we'll need a way to mark the start and end of each cycle or phase of a graph. This at least applies to the polynomial and sine functions, and not to the oscillating stochastic one. In order to find the phase of the first two functions, we'll need to determine the x intercepts of each one. There's a mathematical way of determining both. However, we'll use a different algorithmic approach that employs Clojure's data traversal mechanisms. We can also randomize phase entry points and the sample length taken from each phase.

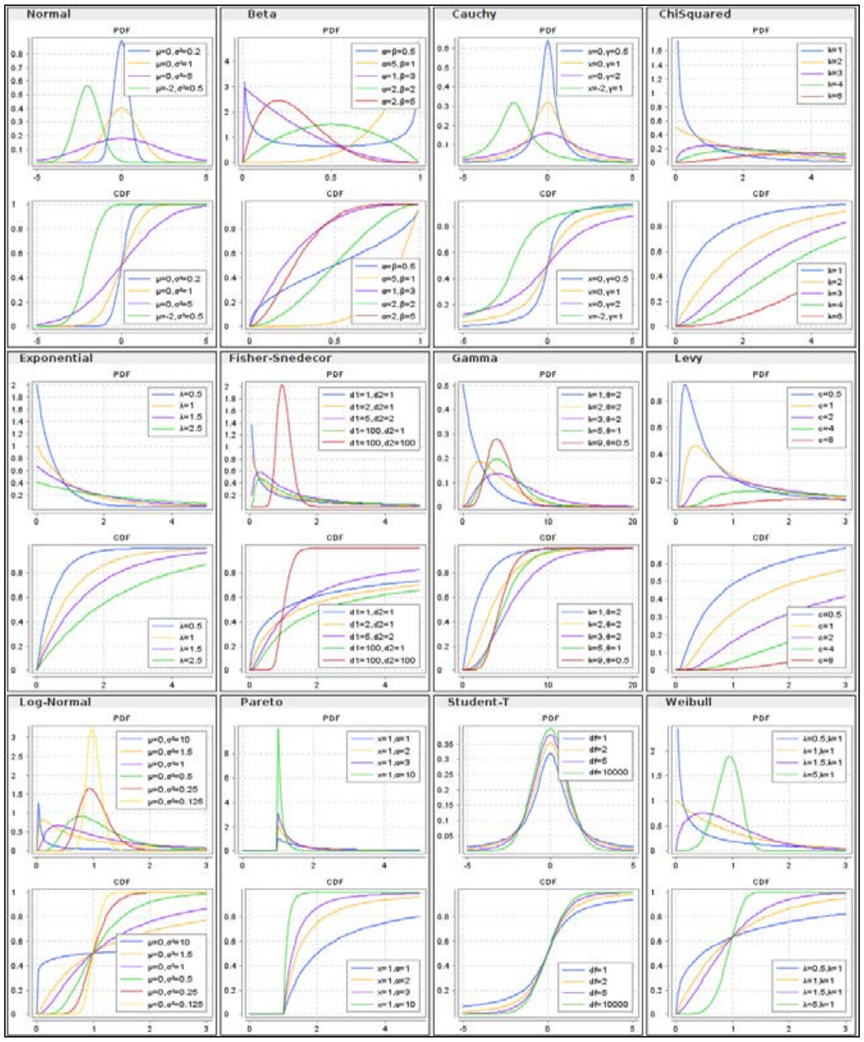
Since we're using these math functions to mimic the behavior of real stock graphs, we'll want to randomize a few other inputs to each graph.

We can randomize the vertical dilation of both the sine and polynomial graphs. For the sine wave, this is done by adjusting the value of the amplitude (or a) of $y = a \sin (b(x - Pi/2)) + d$. A larger amplitude increases the vertical dilation. For the polynomial, the vertical (and, in some cases, horizontal) dilation are controlled by the a variable. However, in this case, larger values result in smaller curves. For example, *(0.5 * x^3) + (b * x^2) - (c * x)* yields a larger curve than *(2 * x^3) + (b * x^2) - (c * x)*.

We randomize the horizontal dilation of both graphs. The sine wave's horizontal dilation is controlled by the b variable in $y = a \sin (b(x - Pi/2)) + d$. The larger the absolute value of b, the tighter each cycle is, that is, 2 yields tighter cycles than 1, and -2 yields tighter cycles than -1. The polynomial wave's horizontal (and, in some cases, vertical) dilation is controlled by its b variable in *(0.5 * x^3) + (b * x^2) - (c * x)*.

You may remember how taking y values at an increment of 1 of each whole number gave a very coarse view of a curve. When we zoomed into the steps at each 1/10th increment, we saw a much more granular view of this curve. Accordingly, we can randomize the granularity (or zoom level) of points on both curves.

Finally, we need a way to randomly generate either of our three graph functions with a given set of inputs. Clojure lets us randomly select integers, so we can take a random integer within a range, determine if it's within a subrange, then sample (that is, generate) a graph from it. However, math also gives us useful tools. In this case, a probability distribution can help us evenly distribute graph generation according to inputs for a given probability curve. There are a number of probability (or continuous) distributions available (normal, exponential, beta, gamma, and so on). A graphic from Apache Commons math library gives an overview of some probability distributions, which can be seen at `https://en.wikipedia.org/wiki/Probability_distribution`:



Source: http://commons.apache.org/proper/commons-math/images/userguide/real_distribution_examples.png

We're going to use a beta distribution for our function. Try out an online beta distribution calculator at `http://keisan.casio.com/exec/system/1180573226`. Plug in the initial values of *a*=2 and *b*=3 and your result. The idea is that if we take a sample of 100 items, the likelihood that a value will be under 0.33 is determined by the shape of the curve. Therefore, there is a likelihood that a sample will be between 0.34 and 0.66 or above 0.67. We're going to use these very metrics to make online decisions of which price graphs get generated. By online, I mean we do not have to hardcode our sections between 0 and 1 [for example, 0.66 and 0.86 (0.20); the remainder will be (0.14)]. We can evenly divide an area between 0 and 1, and if the hump of our probability curve is skewed to the left-hand side of the graph, there's a greater probability that sampled values will fall under this area of the graph.

Surveying the function landscape

In order to accomplish everything we've spoken about so far, let's take a step back and explore some functions to traverse data, branching, and conditional dispatch, and other first order functions. The Clojure cheatsheet is a good location for a bird's-eye view of different categories of functions. Their official documentation can be found at `http://clojure.org/cheatsheet`.

Traversing data

Recursion in computer science is a technique where a function continuously calls itself until some decision point is reached (you can read more about this technique at `https://en.wikipedia.org/wiki/Recursion_(computer_science)`). This is useful when the same block of code needs to be run against successive or iterative contexts.

The `reduce` function is used when we have a finite list that we need to collect or fold up into a result value.

Clojure also has the `repeatedly` function, which continuously calls a function with no arguments. There's also `iterate`, which is a single argument function that takes its results and successively calls the same function on successive results.

These options constrain the number and evaluation of input arguments. If more input arguments are needed, you also have the option of simply recalling your same function over and over again until some condition is reached. However, this has some internal side effects where a resource called stack is consumed. The `loop` function (technically a macro) lets us recur over a block of code without consuming this stack. It also has the benefit of allowing us to set a different recursion point in our code. Paired with `loop` is the `recur` function (also a macro) that evaluates a set of expressions, goes back to the recursion point (set by the preceding `loop` function), and binds these results in a source binding location. The `recur` operator can also be used outside `loop` where it will just call the available recursion point, which would be the function itself.

List comprehensions in computer science are ways of creating new lists from previous lists. The `map`, `filter`, and so on, are the functions which generally let us create a new list from a source list using one condition or pass over the source data. Clojure's `for` comprehension lets us take an arbitrary number of lists and create a new result list based on one or more conditionals or modifiers.

Branching and conditional dispatch

The macros, `case` and `cond`, are two straightforward ways of making a decision. `case` is a macro that looks at one value (scalar or composite) and tries to find a matching pair in the following body. This is a good option when there's a single known Var and we want to take different actions based on its value. `cond`, on the other hand, is a macro that simply tries to find a true condition in a series of pairs. It can be used when you need to meet a condition between several competing predicate values.

Multimethods let us add decision points (or dispatches) in our code based on any attribute of a value, such as its type, metadata, and so on. This functionality, called **runtime polymorphism**, is provided by Clojure's `defmulti` and `defmethod` macros.

Pattern matching in computer science is a language feature that lets us check a set of values for a particular pattern. Dispatch of a particular kind of code occurs when a pattern is matched. Clojure's `core.match` is a third-party library that implements pattern matching as an add-on to the language. This, again, is made possible by macros, which rewrite the code on the fly. It's another example of a little language being embedded into the main one.

First order functions

The central artifact of a functional programming language is the function, also called a **lambda**. It is the primary way of doing things and is itself a value that can be manipulated in the given language. Clojure has many functions that in turn operate on or change other functions. Here, we will discuss a few.

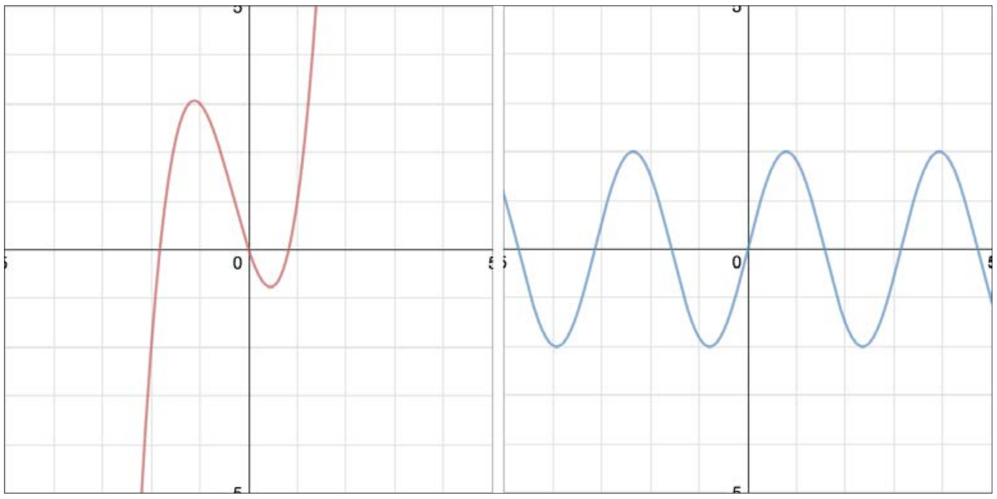
The `partial` function takes a function of any number of arguments. What it does is apply the function, but only to a partial set of arguments. A new function is returned, which is a partial application of the original. So, a caller now just has to call the function with the remaining arguments. There are some instances in your code where you will want to partially apply a function in one part and will only be able to complete the function application in another location or later on. The `apply` function takes another function and applies it over a list of arguments. Its need is motivated by the fact that we sometimes don't have access to all arguments to a function when writing and compiling (part of the REPL) our code. Only at runtime can we access a function's arguments. This is the main motivation for its use. The `comp` is a function that takes several functions and composes them together. This is often useful when several functions are used together in the same place. We can create a more general function out of its composition and use it in turn. The `juxt` function takes a set of functions and returns a function that is a juxtaposition of these very functions to input arguments of the original `juxt` call. This is useful when you have a set of functions that you want to call on the same argument.

Applying functions

Let's make a function to determine phases or the start and end of our graph's cycles. We'll write a function that finds the `x` intercepts of both the polynomial and sine graphs. Recall that these two functions appear as follows:

```
(defn polynomial [a b c x]
  (->
   (+ (* a
         (Math/pow x 3))

      (* b
         (Math/pow x 2)))
   (- (* c x))))

(defn sine [a b d x]
  (- (* a
        (Math/sin (* b
                     (- x
                        (/ Math/PI 2)))))
     d))
```

Looking again at the representative graphs, the first thing to note is a constant *x* intercept of *x=0*. Next, if we go in either direction, the graph goes in a particular direction, then returns to pass through *x*, and winds back up again:



The graph on left-hand side shows a polynomial curve and the one on right-hand side shows a sine wave

Algorithmically, we can use this to first determine the direction (or side) of the curve. Next, when we pass through `x`, the direction (or side) will change. If the side does change, we need to go back in the other direction until the side changes again. We can keep doing this in increasingly smaller steps until we have an effective 0 value. For our purposes, I've chosen an `x` of 0 to the 7th power. Any values beyond this can be considered negligible to our phase.

To implement our x-intercept function, we'll need to track a few variables. At each step, we need to know the following:

- The starting point
- The distance to step
- The direction (or side) of the curve
- The corresponding *y* value, which will ultimately get returned when *x* is effectively 0

With all these variables (or states) to track, `loop/recur` looks to be a useful function. We can track multiple values and recursively evaluate them until we reach our desired result:

```
(defn polynomial-xintercept [x]
  (polynomial 2 2 3 x))
```

```
(defn sine-xintercept [x]
  (sine 2 2 0 x))

(defn ydirection [ypos]
  (if (pos? ypos)
    :positive
    :negative))

(defn direction-changed? [ypos dirn]
  (not (= (ydirection ypos)
          dirn)))

(defn get-opposite-direction-key [ydir]
  (if (= ydir :positive)
    :negative
    :positive))

(defn get-opposite-direction-fn [dirn]
  (if (= dirn +) - +))

(defn effectively-zero? [xval]
  (= 0.0 (Double. (format "%.7f" xval))))

(defn find-xintercept [direction mfn]
  (loop [start-point 0.0
         distance    1.0
         ydir        (ydirection (mfn (direction 0 0.1)))
         dirn        direction]

    (let [next-point (dirn start-point distance)]

      (if (effectively-zero? (mfn next-point))
        next-point
        (let [dc? (direction-changed? (mfn next-point) ydir)]

          (recur next-point
                 (if dc? (/ distance 2) distance)
                 (if dc? (get-opposite-direction-key ydir) ydir)
                 (if dc? (get-opposite-direction-fn dirn) dirn))))))))
```

So, this is one implementation that works. Focus on the `loop` and `recur` bindings. The `let` and `if` functions revolve around setting up and deciding whether to continue. We have some helper functions that let us split code. These make some of the bits of code reusable and the main function more readable. The x-intercept algorithm basically follows these steps:

1. Sets up an initial *y* value.
2. Calls `ydirection` to determine the graph's direction (or side). This is where we set up and use the `sine-xintercept` and `polynomial-xintercept` helper functions. They simplify the parameters passed to them.
3. Determines if *x* is effectively 0. This code is split out into the `effectively-zero?` helper function:
    - If it is effectively 0, return the associated *y* value
    - If it's not 0, check whether the direction (or side) of the curve has changed (`direction-changed?`)
        - If it has, recurse with i) the *y* value, ii) half the distance to a step, iii) a flag marking the opposite direction, and iv) a function for the opposite direction
        - If it has not, recurse with i) the *y* value, ii) the same step distance, iii) the same direction flag, and iv) the same step function

Now, if we want to find the polynomial's or sine's x-intercept, we can use `find-xintercept` as follows. Also, note that we can find x-intercepts on both the left and right-hand side of *x=0*:

```
(find-xintercept - polynomial-xintercept)
-1.8228756561875343

(find-xintercept + polynomial-xintercept)
0.8228756487369537

(find-xintercept - sine-xintercept)
-1.570796325802803

(find-xintercept + sine-xintercept)
1.570796325802803
```

In order to randomize vertical and horizontal dilations, we'll need a function that generates a random double value within a specified range. The Clojure `core` namespace doesn't have such a function. However, we can use one from an older project, lazytest, which is no longer under active development. The following code is taken from `https://github.com/stuartsierra/lazytest/blob/master/modules/lazytest/src/main/clojure/lazytest/random.clj`:

```
(defn rand-double-in-range
  "Returns a random double between min and max."
  [min max]
  {:pre [(<= min max)]}
  (+ min (* (- max min) (Math/random))))
```

Taking a first pass at randomizing the vertical dilation for the polynomial and sine equations, we get the following functions:

```
(defn randomize-vertical-dilation-P [x]
  (let [a (rand-double-in-range 0.5 2)]
    (polynomial a 2 3 x)))

(defn randomize-vertical-dilation-S [x]
  (let [a (rand-double-in-range 0.5 2.7)]
    (sine a 2 0 x)))
```

However, both shapes (polynomial and sine) are very similar. These can, therefore, be generalized into one function called `randomize-vertical-dilation`. With this function derived, we can reuse a pattern to create `randomize-horizontal-dilation`. Also, notice how adjusting the vertical and horizontal dilations means adjusting the `a` and `b` values as well:

```
(defn randomize-vertical-dilation [mathfn min' max']
  (let [a (rand-double-in-range min' max')]
    (partial mathfn a)))

(defn randomize-horizontal-dilation [mathfn-curried min' max']
  (let [b (rand-double-in-range min' max')]
    (partial mathfn-curried b)))
```

This is a good use case to partially apply a function with the first argument (`a` value). Take this partially applied function and then partially apply it again to `b` and any other successive values. We'll then find x-intercepts, select a granularity, and then iteratively generate *x* values by stepping (`iterate`) forward at each granularity increment, starting with the left-most x-intercept. Finally, we can map our partially applied polynomial or sine functions over the granular sequence of x's, yielding y's at each of these *x* points.

For each feature, we're selecting from a range of random values that work in this context. You can play with these numbers and try whatever works for you:

Vertical dilation:

- **polynomial**: We've selected values within 2 and 0.5
- **sine**: We've selected values between 0.5 and 2.7

Horizontal dilation:

- **polynomial**: We've selected values within 2 and 0.5 (a larger `b` yields a narrower curve)
- **sine**: We've selected values within 2.7 and 0.3

Granularity:

- **polynomial**: We've selected values between 0.1 and 1
- **sine**: We've selected values between 0.1 and 1

Let's take a look at the following code which demonstrates the implementation of what we just covered:

```
(def one (randomize-vertical-dilation polynomial 0.5 2))
(def two (randomize-horizontal-dilation one 0.5 2))
(def polyn-partial (partial two 3))

(def xinterc-polyn-left (find-xintercept - polynomial-xintercept))
(def xinterc-polyn-right (find-xintercept + polynomial-xintercept))

(def granularityP (rand-double-in-range 0.1 1))
(def xsequenceP (iterate (partial + granularityP) xinterc-polyn-left))

(map polyn-partial xsequenceP)
```

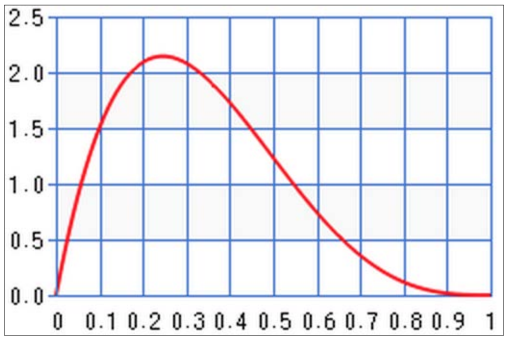
Our sequence generation functions now look like this:

```
(defn generate-polynomial-sequence []

  (let [one (randomize-vertical-dilation polynomial 0.5 2)
        two (randomize-horizontal-dilation one 0.5 2)
        polyn-partial (partial two 3)

        xinterc-polyn-left (find-xintercept - polynomial-xintercept)
        xinterc-polyn-right (find-xintercept + polynomial-xintercept)

        granularityP (rand-double-in-range 0.1 1)
```

```
          xsequenceP (iterate (partial + granularityP) xinterc-polyn-left)]

      (map polyn-partial xsequenceP)))

  (defn generate-sine-sequence []

    (let [ein (randomize-vertical-dilation sine 0.5 2.7)
          zwei (randomize-horizontal-dilation ein 0.3 2.7)
          sine-partial (partial zwei 0)

          xinterc-sine-left (find-xintercept - sine-xintercept)
          xinterc-sine-right (find-xintercept + sine-xintercept)

          granularityS (rand-double-in-range 0.1 1)
         xsequenceS (iterate (partial + granularityS) xinterc-sine-left)]

      (map sine-partial xsequenceS)))

  (defn generate-oscillating-sequence []
    (analytics/generate-prices-without-population 5 15))
```

Apache Commons Math is a Java library that implements a Beta probability curve. Clojure is a language that runs atop the JVM and can leverage this library for its own use. Remember that we now want to combine all our price generating functions into one long stream of price data. This means that we can distribute the polynomial, sine, and stochastic oscillating functions under a beta curve.

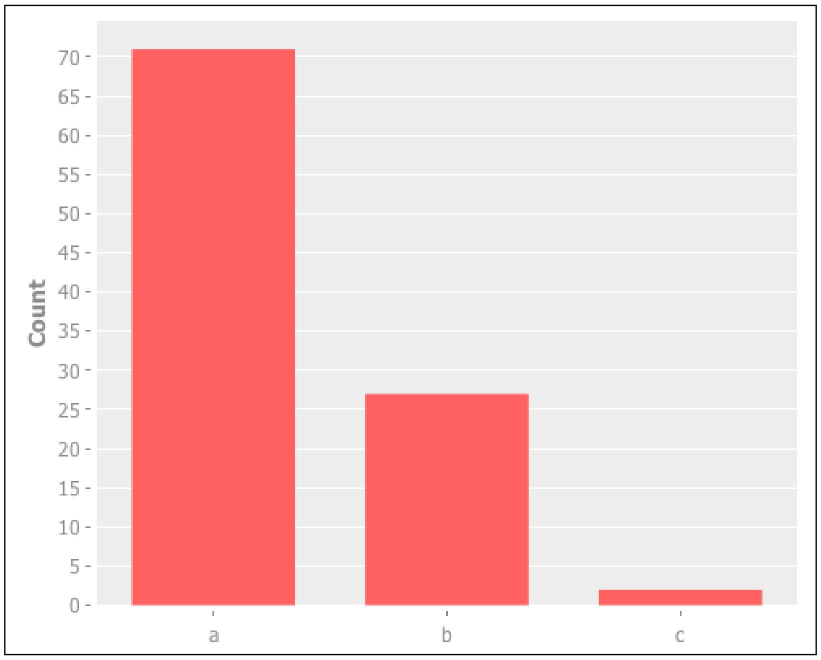
If we create a beta distribution of *a=2*, *b=4.1*, and *x=0* (implicit in the library), the curve looks like this:



Using this, we'll sample evenly in 1/3^{rds} from the pool of our price functions. If we sample from the pool 100 times, the probability distribution will determine how often each third gets sampled. So, for example, in your REPL, try calling `test-beta` 100 times, which will yield the following results (sorted):

```
(defn test-beta [beta-distribution]
  (let [sample-val (.sample beta-distribution)]
    (cond
     (< sample-val 0.33) :a
     (< sample-val 0.66) :b
     :else :c)))

(def beta-distribution     (org.apache.commons.math3.distribution.BetaDistribution. 2.0 4.1))

(def result (repeatedly #(test-beta beta-distribution)))
(sort (take 100 result))

'(:a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a
:a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a :a
:a :a :a :a :a :a :a :a :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b
:b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b :b
:b :b :b :c :c :c :c :c :c)
```

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



With this in mind, it's now trivial to create a simple function that samples from the beta distribution. If a sample falls within a specified range, generate a sequence with a random length (itself within a given range):

```
(defn sample-dispatcher [sample-type sample-length sample-fn]
  (take sample-length (sample-fn)))

(defn sample-prices [beta-distribution]

    (let [sample-val (.sample beta-distribution)]

    (cond
    (< sample-val 0.33) (sample-dispatcher :sine (rand-double-in-range 10 15) generate-sine-sequence)
    (< sample-val 0.66) (sample-dispatcher :polynomial (rand-double-in-range 4 6) generate-polynomial-sequence)
    :else  (sample-dispatcher :oscillating (rand-double-in-range 8 10) generate-oscillating-sequence))))
```

We can now generate an infinite sequence of price samples. The `(repeatedly #(sample-prices beta-distribution))` expression will give us a list of lists, each sublist being a sample of either a polynomial graph, sine graph, or a sequence of oscillating stochastic points. This is where we'll begin the next step. We could just `concat` them all together (recall that `concat` acts lazily) and be done with it. However, the last problem is that each list has its own *y* starting point. For the entire sequence of prices to make sense, we have to normalize the price levels of each subsequent sequence:

```
(defn generate-prices [beta-distribution]

  (reduce (fn [^clojure.lang.LazySeq rslt
               ^clojure.lang.LazySeq each-sample-seq]

            (let [beginning-price (if (empty? rslt)
                                    (rand-double-in-range 5 15)
                                    (last rslt))
                  sample-seq-head (first each-sample-seq)
                  price-difference (math/abs (- sample-seq-head beginning-price))]

              (if (< sample-seq-head beginning-price)
              (concat rslt (map #(+ % price-difference) each-sample-seq))
            (concat rslt (map #(- % price-difference) each-sample-seq) each-sample-seq))))
          '()
          (repeatedly #(sample-prices beta-distribution))))
```

You're probably familiar with Clojure's basic functions by now. And your first instinct may be to `reduce` over the infinite sequence and adjust all the prices of a subsequent list based on the last price of the previous list. However, this approach will fail. The `reduce` function is a core Clojure function that does not act lazily. It is meant to be used when we have finished accumulating data into a list. It then folds over all the elements in a list to arrive at a final value. This means realizing any lazy values. So, if we have an infinite sequence, the preceding function will run indefinitely, which is what `generate-prices` does. If we run `(generate-prices beta-distribution)`, it will never return a list of generated prices as it will never finish reducing. Clearly, we need another approach:

```
(defn generate-prices-iterate [beta-distribution]

  (let [sample-seq (repeatedly #(sample-prices beta-distribution))

        iterfn (fn [[^clojure.lang.LazySeq rslt
                     ^clojure.lang.LazySeq remaining-sample-seq]]

                 (let [each-sample-seq (first remaining-sample-seq)
                       beginning-price (if (empty? rslt)
                                         (rand-double-in-range 5 15)
                                         (last rslt))
                       sample-seq-head (first each-sample-seq)
                  price-difference (math/abs (- sample-seq-head beginning-price))]

                  ;; only raise the price if below the beginning price
                   (if (< sample-seq-head beginning-price)

                 [(concat rslt (map #(+ % price-difference) each-sample-seq))
                      (rest remaining-sample-seq)]
                 [(concat rslt (map #(- % price-difference) each-sample-seq))
                      (rest remaining-sample-seq)])))]

    (map first (iterate iterfn ['() sample-seq]))))
```

Instead of `reduce`, we can use `iterate` to successively call a function and only one argument. This works. However, because we need to maintain the state between lists, we end up having to manually maintain the results and remaining lists between calls. This result in code that is very unclear and the code is unintuitive as a result of having to handle `rslt` and `remaining-sample-seq` (Var bindings in the preceding code):

```
(defn generate-prices-for [beta-distribution]

  (def previous-price nil)

  (let [adjusted-samples (for [each-sample-seq (repeatedly #(sample-prices beta-distribution))
                                    :let [beginning-price (if (nil? previous-price)
                                                              (rand-double-in-range 5 15)
                                                                  previous-price)

                                                  sample-seq-head (first each-sample-seq)
                                   price-difference (math/abs (- sample-seq-head beginning-price))

                                      adjusted-sample (if (< sample-seq-head beginning-price)
                                                          (map #(+ % price-difference) each-sample-seq)
                                                          (map #(- % price-difference) each-sample-seq))

                                   _ (alter-var-root #'previous-price (fn [x] (last adjusted-sample)))]]

                                    adjusted-sample)]

        (apply concat adjusted-samples)))
```

So, we need to remain lazy while cleanly handling intermediate state that is required by a list and its predecessor. The `for` function (a macro) is Clojure's way of implementing list comprehensions. It lets us manipulate many lists at a time in order to create a single sublist. Again, `(repeatedly #(sample-prices beta-distribution))` creates a list of lists, which will be the input to our `for` comprehension. Each sublist is the focus for the `each-sample-seq` binding and the `:let` modifier associated with it. It can, along with the `:while` and `:when` modifiers, apply conditions for how to modify or consider each value in the list. This means that there's much less state for us to maintain. We can simply adjust `each-sample-seq` based on whether the first price is above or below the last price. While this approach is lazy, we have once again not been able to avoid a little hack to maintain the state of `previous-price`:

```
(defn generate-prices-partition [beta-distribution]

  (let [samples-sequence (repeatedly #(sample-prices beta-
distribution))
        partitioned-sequences (partition 2 1 samples-sequence)
        mapping-fn (fn [[fst snd]]
                     (let [beginning-price (last fst)
                           sample-seq-head (first snd)
                   price-difference (math/abs (- sample-seq-head beginning-price))]
                       (if (< sample-seq-head beginning-price)
                         (concat fst (map #(+ % price-difference) snd))
                         (concat fst (map #(- % price-difference) snd)))))]
    (apply concat (map mapping-fn partitioned-sequences))))
```

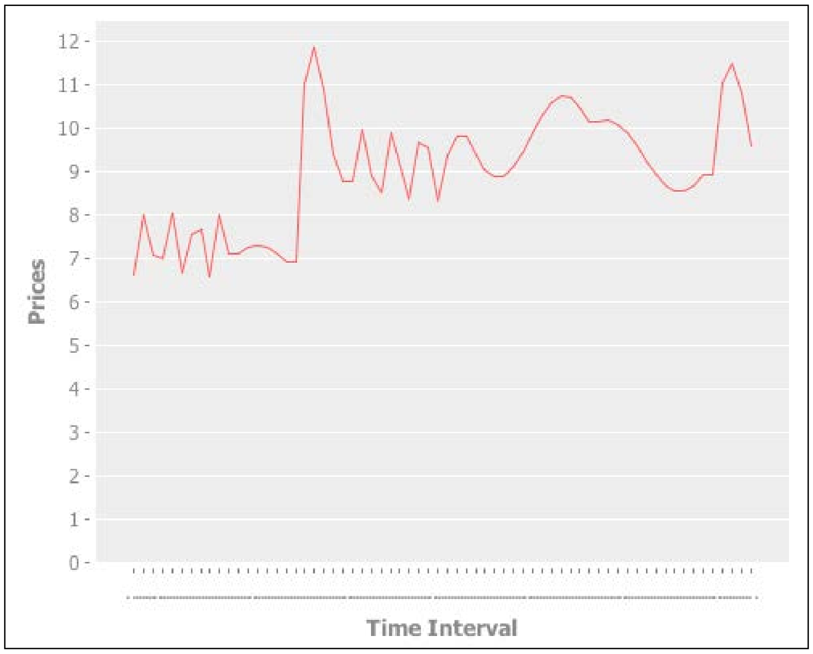
The `partition` function is a good idea as it lets us divide our list of lists into successive pairs (incremented by 1). But, each modified list can't be seen by the successive pair. So, `generate-prices-partition`, while much cleaner, produces a price stream with large and arbitrary price jumps:

```
(defn generate-prices-reductions [beta-distribution]

  (reductions (fn [^clojure.lang.LazySeq rslt
                   ^clojure.lang.LazySeq each-sample-seq]

                (let [beginning-price (if (empty? rslt)
                                        (rand-double-in-range 5 15)
                                        (last rslt))
                      sample-seq-head (first each-sample-seq)
                      price-difference (math/abs (- sample-seq-head beginning-price))]

                  ;; only raise the price if below the beginning price
                  (if (< sample-seq-head beginning-price)
                    (concat rslt (map #(+ % price-difference) each-sample-seq))
                    (concat rslt (map #(- % price-difference) each-sample-seq) each-sample-seq))))
              '()
              (repeatedly #(sample-prices beta-distribution))))
```

Finally, we arrive at a way of maintaining successive states lazily through our infinite sequence of lists. The `reductions` function produces a lazy sequence of all the intermediate values of the `reduce` function. More importantly, however, is the fact that during function processing, previous intermediate results are available to the current iteration. We now have a clean solution that is very close to our original `reduce` version. We've also had a chance to see the utility and trade-offs of the `partition`, `for`, and `iterate` functions in this situation. We can now stitch these into an updated `generate-prices` function with a simple guard condition to ensure that our prices aren't negative:

```
(defn generate-prices
  ([] (generate-prices (BetaDistribution. 2.0 4.1)))
  ([beta-distribution]
   (map (fn [x]
          (if (neg? x) (* -1 x) x))
        (distinct
         (apply concat (generate-prices-reductions beta-distribution))))))
```

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



Summary

When refactoring the `generate-prices` function, we took a step back and considered some useful math equations that could help us generate a list of prices. Indeed, the central moral of Clojure is that it should be a building material or simply the building blocks we use when engineering our systems. We were able to confidently combine results from polynomial and sine equations to our pre-existing stochastic oscillating function.

For the new functions, we were able to vary the vertical dilation, horizontal dilation, and granularity of the samples we took. We were also able to combine these results within a beta probability distribution that we configured to our needs. Much of the work was this kind of math-based consideration. Fleshing out the overall approach of Clojure's functions around recursion, list comprehensions, conditional dispatch, and first order functions then enabled us to traverse and manipulate our data until we got what we wanted.

Dwelling on the different approaches of `reduce`, `iterate`, `for`, and `partition` provided us with a better intuition on how Clojure approaches list transformation. We've only touched on the functionality available for data and program transformations. In the next chapter, we'll step back and review all of Clojure's features together.

6
Surveying the Landscape

So far, this book has been driven by the problem domain, pulling in Clojure's features as we need to solve a specific problem. The purpose of this chapter is to step back and review all of Clojure's features together. We'll review Clojure's scalars, collection types, and composite data types again. Then, we'll take a look at how to use Clojure's functions to access, update, and compose data structures.

We'll look at I/O operations and ways to approach what are called side effects. We'll also touch on Clojure's options when dealing with concurrency and parallelism and gain an understanding of the difference between the two. We'll then take our accumulated Clojure knowledge and compare it to similar computation models offered by object-oriented and strongly typed functional programming systems using Java and Haskell, respectively. This will be a good opportunity to gain a basic understanding of type systems, how they apply to Clojure, Java, and Haskell, and the implications of each computation model. The topics that we'll cover in this chapter are as follows:

- Scalar data types
- Numbers and precision
- A review of collections
- Data transformation patterns and principles
- Clojure's model of state and identity
- Introducing side-effects
- Concurrency and parallelism
- Type systems
- Comparing Clojure with object orientation
- Comparing Clojure with FP and strong typing

Scalar data types

Scalar data types are those that contain one value at a time, as opposed to collections of composite data types. A scalar can be in the form of a nil, boolean, number, string, character, regular expression, keyword, or symbol. The following program elements are all scalar in nature and can be directly evaluated in your REPL:

```
nil
true
4
2.3
567/11
"hello"
\b
#"^hello\d?world$"
:hello
'hello
```

Numbers and precision

Clojure numbers default to their host implementation, which refers to the JVM in most cases. So, `1` is of type `java.lang.Long`, and `2.5` or `0.1` is of the `java.lang.Double` type:

```
(type 1)
java.lang.Long

(type 0.1)
java.lang.Double
```

Clojure supports Java's primitive number types of `byte`, `char`, `int`, `long`, and `float`. These symbols also correspond to Clojure functions, which convert to a native type. This is useful when dealing with a function where a particular type hints at a primitive number:

```
(defn double' ^long [^long n]
  (+ n n))

(double' (int 4))
8
```

Clojure also supports native JVM numeric types, such as Byte, Short, Integer, Long, BigInteger, Float, Double, and BigDecimal. Larger numbers in Clojure have custom types because they require a greater internal memory representation or a different method of processing. The `clojure.lang.BigInt` class is used whenever we need to represent integers that are larger than 64 bits. It is also used for rational numbers that can't be represented with terminating decimals:

```
(type 1000000000000000000000)
clojure.lang.BigInt

(* 2.9 10000000000000000000000)
2.8999999999999998E22

(type (* 2.9 10000000000000000000000))
java.lang.Double

(type 1/3)
clojure.lang.Ratio

(def a 5/6)

a
5/6

(* a 9)
15/2

(* 2 7.936e-16)
1.5872E-15
```

Math operations work seamlessly both on converting values as necessary and maintaining their precision. However, this only applies when the numbers fit within Clojure's numeric types and default memory sizes (or when performing an interoperation between Clojure and Java). If you fall outside these memory ranges, your numbers might experience truncation, promotion, overflow, underflow, or rounding errors (you can read more at `http://dev.clojure.org/display/design/Documentation+for+1.3+Numerics`):

- Truncation occurs when the precision of a floating point number is limited because we can't fit the necessary bits into a particular representation.
- Promotion occurs when a number is too large for its representation. Clojure will automatically promote this number to a bigger type. For example, promoting `java.lang.Double` to `java.lang.BigDecimal` or `java.lang.Integer` to `java.lang.Long`.

- Overflow mostly takes place by operating on `int` and `long` primitive values when the result is larger than its 32 bit representation.
- Underflow is the opposite, occurring when a number is so small that its value is set to zero.
- Rounding errors occur when a floating number type isn't big enough to handle the actual value that's needed. This often happens when interacting with Java libraries and is very hard to spot because the error is small. These small errors, though, can accumulate over time and negatively affect your calculations. So, pay close attention to your numbers and your calculations over time.

In *The Joy Of Clojure*, by Manning, Michael Fogus describes how to evade preceding the conditions when you need to (you can read more at `https://www.joyofclojure.com/`). This is done using Clojure's `clojure.lang.Ratio` type, (that is, the Rational Type) to attain perfect accuracy in your calculations. The `ratio?`, `rational?`, and `rationalize` functions help you determine whether your number(s) are rational and convert them to a rational type (or ratio) if the need arises. Fogus recommends that with really large or really small numbers, if you need to maintain precision, check the associative and distributive properties of the numbers resulting from your calculations. By **associative**, we mean that it doesn't matter how we group our numbers:

```
(def a (rationalize 1.0e50))
(def b (rationalize -1.0e50))
(def c (rationalize 17.0e00))

(+ (+ a b) c)
17

(+ a (+ b c))
17
```

A **distributive** property signals an indifference to how we distribute a multiplicative factor. With the help of the following code, we can multiply `a` by the sum of `b` and `c`. Or, we can multiply `a` by `b`, and then add it to `a` times `c`. Both should yield the same result:

```
(let [a (rationalize 0.1)
      b (rationalize 0.2)
      c (rationalize 0.3)]

  (= (* a (+ b c))
     (+ (* a b) (* a c))))
true
```

A review of collections

We introduced the Clojure collection types in *Chapter 1*, *Orientation – Addressing the Questions Clojure Answers*, and *Chapter 3*, *Developing the Simple Moving Average*. They are all immutable with sequences having the extra facility of laziness. Let's review what we know so far:

- **Lists**: These are data structures that are surrounded by the `()` parentheses. Unquoted, lists evaluate functions in a list's head position `(myfn 1 2 3)`. Any number of arguments can follow after the head. Quoted, lists are simply data. They can contain any value type `'(1 2 3 myfn :fubar)`.
- **Vectors**: These are data structures with efficient random access, where list's efficient point of access is its head. Vectors are written with two opposing square brackets, for example, `[75 6 452 40]`.
- **Sets**: These are collections of unique items where only one of each thing can be put into a set. They are written with curly braces and prefixed with a hash, for example, `#{1 2 3 4}`. A set can also act as a "does this exist" function of its items. So, `(#{"a" "s" "d" "f"} 2)` will return `nil` and `(#{"a" "s" "d" "f"} "d")`, will return the `d` item.
- **Maps**: These are a part of Clojure's associative data structures, associating keys to values. They are denoted by opposing curly braces, such as `{:hello "world" :foo "bar"}`. A map entry is a pair of keys and values. Keys and values can refer to any Clojure data structure. However, very often, you'll see Clojure code using keywords as keys. Like sets and keywords, maps act as functions, performing an associative lookup of their elements. So, evaluating `({:hello "world" :foo "bar"} :foo)` will return `bar`.
- **Sequences**: A sequence, similar to a list, is a linked list of items. Sequences are also written with parentheses, but have the additional property of laziness. A lazy sequence does not immediately compute any value inside itself. This is a central notion in functional programming, leading to more performant systems. Only when a value is consumed by another non-lazy process, will the value and, crucially, the computer's resources be consumed. This is a central part of Clojure's computation model, allowing us to stitch together unrealized results of lazy functions and creating a kind of data flow pipeline. Strategically, we'd prefer to execute as much of our data processing in this manner, thus conserving our compute resources and letting us swap in and out transformation functions that consume and produce lazy sequences. Sequences allow loose coupling (`https://en.wikipedia.org/wiki/Loose_coupling`) between functions and components, allowing data to drive design.

- **Queues**: Persistent queues are Clojure's **First-in First-out** (**FiFo**) implementation. These don't have a reader-literal syntax, only allowing empty queue creation with `clojure.lang.PersistentQueue/EMPTY` (for example, `(def q1 clojure.lang.PersistentQueue/EMPTY)`). A persistent queue is immutable, thus functions like `conj`, `peek`, and `pop` return new values, leaving the original value intact.

Data transformation patterns and principles

Clojure's collection types are, by definition, composite as they are composed of many elements. However, we can also put collections within each other, nesting these data structures to fit your needs. As such, we'd like to know the best approaches to build composite collections, access, and update collections. We'll also look at patterns of grouping, sorting, mapping, filtering, and reducing our data, as well as avoiding mutations. We'll also take a look at how to introduce side effects when necessary. We're going to return to the Clojure cheatsheet (`http://clojure.org/cheatsheet`) and look at some of the basic functions needed to do the following. Try evaluating these functions in your REPL:

- **Accessing and updating collections**: These are just some of the functions that let you get and update values inside your data structures:
    - `get`: This returns a value mapped to a key such as `(get [1 2 3] 1)` `(get {:a :s :d :f} :s)`.
    - `get-in`: This returns a value that's nested in an associative data structure such as `(get-in {:a [1 5] :b [10 20]} [:b 1])`.
    - `update`: This updates a value mapped to a key in an associative data structure such as `(update {:one 1 :two 2} :two inc)` `(update {:a "fu" :b "world"} :b (fn [x] (str "hello " x)))`.
    - `update-in`: This updates a nested value according to a supplied path such as `(update-in {:a [1 5] :b [10 20]} [:b 1] inc)`.

- **Building composite collections**: Building nested and composite data structures very much depends on the combination of the structure of your data and the operation you want to perform on it. For example, our nominal simple-moving-average function can return a list of average prices plus the set of original prices on which this average is based. If there's no processing needed for the original population data, it doesn't need to be included in the simple moving average result.
- **Grouping**: These are functions that let us group elements in a collection:
    - `group-by`: This groups a list of elements by a supplied function. The result is a map where the key is the result of the function and the value is a vector of similar elements, for example, `(group-by :id [{:id 3 :a 1 :b 2} {:id 1 :c 3 :a 4} {:id 3 :a 5 :e 6 :b 7}])`.
- **Sorting**: These are functions that let us sort elements in our collection:
    - `sort`: This returns a lazy sequence that's sorted by a source's natural order or supplied comparator, for example, `(sort [85 623 6 47 2 20]) (sort > [85 623 6 47 2 20])`.
    - `sort-by`: This returns a lazy sequence that's sorted by a supplied key function and, optionally, a comparator function, for example, `(sort-by :id [{:id 3 :a 1 :b 2} {:id 1 :c 3 :a 4} {:id 3 :a 5 :e 6 :b 7}]) (sort-by :id > [{:id 3 :a 1 :b 2} {:id 1 :c 3 :a 4} {:id 3 :a 5 :e 6 :b 7}])`.
    - `sorted-set`: This returns a sorted set of parameters, for example, `(sorted-set 8 9 7 7 8 15 9)`.
    - `sorted-set-by`: This returns a sorted set of parameters using a comparator that's been supplied, for example, `(sorted-set-by > 8 9 7 7 8 15 9)`.
    - `sorted-map`: This returns a sorted map of key value pairs, for example,`(sorted-map "b" 2 "a" 1) (sorted-map :q 1 :w 2 :e 3 :r 4 :t 5 :y 6)`.
    - `sorted-map-by`: This returns a sorted map of key value pairs that are sorted by a comparator function on the keys, for example, `(sorted-map-by > 1 :q 2 :w 3 :e 4 :r 5 :t 6 :y)`.

- **Mapping**: These are functions that let us pass a function over a collection, returning a new collection with the results of a function call on each element each time:
    - `map`: This returns a lazy sequence when applying a function to the first element of each supplied collection, for example, `(map inc [9 8 7 6 5]) (map + [1 2 3 4 5] [6 7 8 9 10])`.
    - `mapv`: This returns a vector when applying a function to the first element of each supplied collection, for example, `(map inc [9 8 7 6 5]) (map + [1 2 3 4 5] [6 7 8 9 10])`.
    - `pmap`: This behaves exactly like `map`, except `pmap` runs a function in parallel, for example, `(map inc [9 8 7 6 5]) (map + [1 2 3 4 5] [6 7 8 9 10])`.
    - `map-indexed`: This returns a lazy sequence of a supplied function where the parameters are the index (starting at 0) and the accompanying element. Try this in your REPL `(map-indexed (fn [idx itm] [idx itm]) [:q :w :e :r :t :y])`.
    - `mapcat`: This maps a function over a list of lists, and then concatenates the result, for example, `(mapcat sort [[5 4 3 2 1] [9 7 8 6]])`.
    - `zipmap`: This returns a map where key/values are derived from an interleaving of two collections, for example, `(zipmap [:q :w :e :r :t :y] [1 2 3 4 5 6])`.
- **Filtering**: These are functions that let us constrain collections by one or more constraints as follows:
    - `filter`: This filters elements and returns a lazy sequence that meets the predicate function's criteria, for example, `(filter keyword? [:a "q" :s "w" :d "e" :f "r" "t" "y"])`.
    - `filterv`: This filters elements and returns a vector that meets the predicate function's criteria, for example, `(filter keyword? [:a "q" :s "w" :d "e" :f "r" "t" "y"])`.
- **Reducing**: These are functions that let us take a list and reduce element values to a single value as follows:
    - `reduce`: Reduce or left fold passes a function over each element, collecting values from left to right, for example, `(reduce + (range 10)) (reduce + 100 (range 10))`.
    - `reductions`: This is like `reduce`, but yields a lazy sequence of intermediate values, of the reduce process. Try this in your REPL `(reductions + [10 20 30 40 50])`.

- `reduced`: This lets you terminate a reduction, returning a passed value:

```
(def a (repeat 1))
(take 100 a)
; => (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)

(reduce (fn[rslt ech]
          (if (< rslt 20)
            (+ rslt ech)
            (reduced :break)))
        a)
; => :break
```

- `reduce-kv`: This lets you iterate over the elements of a map:

```
(reduce-kv (fn [m k v] (assoc m k (inc v)))
           {} {:a 1 :b 2 :c 3}))
 ; => {:a 2, :b 3, :c 4}
```

Clojure's model of state and identity

Clojure has an immutable data model. However, this doesn't mean that we can't represent data and change it over time. Direct change or mutation should be avoided at all costs. If necessary, a `binding` macro will create new `var` values (or bindings) that are local only to a `binding` context or scope. These Vars must be marked as `^:dynamic` and they should already exist:

```
(def ^:dynamic a 1)
(def ^:dynamic b 2)

(binding [a 10 b 20] (+ a b))
; => 30

a
; => 1

b
; => 2
```

It is, however, preferable to choose one of four reference types: Var, Atom, Agent, and Ref. Doing this requires an understanding of Clojure's notion of identity, value, and time. In his presentation *Are We There Yet?* (you can read more at `http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey`), Clojure creator, Rich Hickey, describes most computer languages as having variables that allow programmers to set and change a value in the program's runtime. This model assumes, however, that the world stops while you look at or change it (you can read more at `http://clojure.org/state`). Clojure's approach includes having an immutable value, such as Sally's address, and forever maintaining this value at a point in time. Sally's address, however, is just a value in 2014. Clojure sees identity as a continuous logical entity that has a series of different values over time. So, Sally's home, would be an identity where the value of her address in 2014 may be different from her address in 2015. As we've seen, Clojure's core functions take immutable values and return new values that are updates of the original. The two values share data structures under the hood (Clojure's Persistent Data Structures), which is possible since each value won't change. This succession of data updates is how we disambiguate between values over time. These values often need to be shared with other threads of execution. The Var, Atom, Agent, and Ref reference types are ways of managing shared identities that have a particular state at a given point in time. These reference types can change using a prescribed set of functions to create, update, and reset their state:

- Refs are used for synchronous, coordinated access to many identities. These functions let you modify their values. You can create a ref using `ref`, update it using `alter` or `commute`, and reset it using `ref-set`.

```
(def one (ref []))
     ;; #object[clojure.lang.Ref 0x1fbb15f {:status       :ready, :val []}]

      (dosync
        (alter one conj 12))
      ;; [12]

      (deref one)  ;; [12]
      @one  ;; [12]

      (dosync
        (ref-set one [:fubar]))
      ;; [:fubar]

      @one  ;; [:fubar]
```

- Atoms are used for synchronous, uncoordinated access to a single identity. These functions let you modify their values. You can create an atom using `atom`, update it using `swap!`, and reset it using `reset!`:

```
(def two (atom []))
    ;; #object[clojure.lang.Atom 0xe2fb45 {:status       :ready, :val []}]

      (swap! two conj 12)
      ;; [12]

      @two
      ;; [12]

      (reset! two [:fubar])
         ;; [:fubar]

      @two
      ;; [:fubar]
```

- Agents are used for asynchronous, uncoordinated access to a single identity. These functions let you modify their values. You can create an agent using `agent`, update it using `send` or `send-off`, and reset it using `restart-agent`.

```
(send three conj 12)
    ;; #object[clojure.lang.Agent 0x1198a8c {:status       :ready, :val []}]

      @three
      ;; [12]

      (restart-agent three [:fubar])
      java.lang.RuntimeException
         Agent does not need a restart
         Util.java:  221
         clojure.lang.Util/runtimeException
         Agent.java:  210
         clojure.lang.Agent/restart
         core.clj: 2078
         clojure.core/restart-agent
         RestFn.java:  425
         clojure.lang.RestFn/invoke
         six.clj:   66
         edgaru.six.six/eval14571

      (send three inc)
```

```
java.lang.ClassCastException
 clojure.lang.PersistentVector cannot be cast to         java.lang.Number

(restart-agent three [:fubar])
;; [:fubar]

@three
;; [:fubar]
```

- Vars are used for isolated (or thread local) changes made to identities with a shared default value. These functions let you modify their values. You can create Vars using `def`, update them using `alter-var-root`, and reset them using `var-set`:

```
(def four [])
      ;; []

      (alter-var-root (var four) conj 12)
      ;; [12]

      four
      ;; [12]
```

Introducing side effects

In *Out of the Tar Pit*, Ben Moseley and Peter Marks identify complexity as the single biggest obstacle to the successful development of large-scale software systems (you can read more at `http://shaffner.us/cs/papers/tarpit.pdf`). Essential complexity is directly inherent to solving the problem at hand. Accidental complexity is all that remains and is usually introduced by unnecessary tooling or infrastructure. The authors advocate for avoiding complexity and separating it where unavoidable. Managing our program's identity and making changes to its state should take place in well-contained locations of your code.

The `with-open` macro is an example of how to communicate with the outside world. It takes an input or output stream and binds it to a local `var` that our local expressions can interact with. The complexity here is dealing with an external resource (an input or output stream). The `with-open` macro handles the side effects of reading and writing to outside systems and performs resource cleanup after function completion.

The component pattern, popularized by Stuart Sierra, is widely used in the Clojure community. It isolates and manages the life cycle of units of code (called components) that have runtime state. Managing runtime state is a major area of focus for all programming languages. We've looked at Clojure's basic facilities for data transformation as well as its model of state and identity. More generally, though, we can also consider when you have to perform many simultaneous tasks. Next, we'll look at Clojure's facilities for doing this.

Concurrency and parallelism

Though slightly beyond the scope of this book, there may come a time when you will want to run simultaneous paths of execution. If this happens, it's good to know the options that Clojure provides for running concurrent or parallel code. We should, of course, understand the difference between the two.

Concurrency occurs when many threads of execution are running independent of each other. This can technically happen on a single-core chip where a processor multitasks between performing work in each thread. Concurrency makes resource-intensive software more usable by allowing many things to happen at once. Parallelism is made possible by computer processors with multiple cores. It occurs when multiple tasks physically take place at the same time in a multicore chip. Performance is greatly improved by running many threads of execution at the same time. Clojure provides several ways of tackling concurrent execution based on the needs and constraints of a given situation. These are as follows:

- **Future**: This takes one or more expressions, gives you a future object, and runs this object asynchronously in a separate thread. When complete, the future object returns the result, which you can obtain by dereferencing it.
- **Promise**: This gives you a promise object for which another code block or thread can `deliver` just one result at a later point in time. To see if a promise has been delivered, you can dereference (`deref`) the promise or query it with the `realized?` function.
- **Threads**: The Java language already has an advanced threading library. Java executors (`https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html`), used in larger applications encapsulate thread creation and management into a set of functions called executors.

- **Reducers**: These are a way of executing your functions to perform sequence manipulation, but in a parallel manner. The way this is done and the main rationale for this approach is to separate the what from the how in standard core functions, such as `map` and `reduce`. A reducer combines a reducible collection (one that knows how to reduce itself) with a reducing function (a recipe for what to do during reduction). We won't delve into the details here. Just know that reducers provide a different and more performant way of performing sequence transformations via parallelism (using Java's fork/join framework).
- **core.async**: We've so far seen that Clojure provides many ways of performing concurrency, and parallelism. Another option for concurrency is a pattern known as **Communicating Sequential Processes** (**CSP**). It's technically a formal language in the process algebra family, which is used to describe patterns of interactions in concurrent systems. Clojure's implementation of CSP, also inspired by implementations in the Go language, is called `core.async`. It allows for asynchronous programming using message passing via `channels.Refs` and transactions, agents, and asynchronous actions.

Type systems

Clojure is homoiconic, functional, lazy, and list-based. But, it's also a typed language—dynamically typed in this case (`https://en.wikipedia.org/wiki/Type_system#DYNAMIC`). In computer science, type systems are tools that are mainly used to reduce bugs and improve the correctness of programs. What follows is not an exhaustive discourse on type systems. We only explore their rationale and variations. It's important to know that type systems are just tools that programming languages can employ under a certain set of conditions. There are different categories of type systems: duck typing, dependent types, gradual typing, and so on. In computation, type systems present an approach to data representation and transformation. Haskell, for example, encourages a very strict way of data representation and data transformation. Its type system is very closely derived from category theory (`https://en.wikipedia.org/wiki/Category_theory`), which is a mathematical formalism. Essentially, category theory rigorously tries to take some state and transform it to another state. These are attempts at having a provable system. The focus on types of data and their transformations gives us more clarity on when and why we should use these tools. There's a lot of value in having a system that we can prove is always correct for a given set of concurrent, distributed inputs. These properties are important in many domains: military, finance, global communications, and so on.

To apply these concepts in the real world, let's think in terms of mathematics and consider how acoustic waves get transformed into digital representations. How is video captured, and what algorithms can we run over top of the data to flip it, or get a negative of an image. So, what is an image and how would you represent it digitally, in order to perform some of these interesting transformations? These data representation and transformation questions are what Type Systems are addressing. Type systems are closely related to and informed by type theory (also used in mathematics and logic), which formally studies the different forms of type systems. Type systems try to assure correctness by applying types to program elements (functions, variables, and so on). It may be useful to think of types as a contract between program elements to behave in a certain way. The contracted behavior can be checked at compile time (statically) or runtime (dynamically). Compile time or static checking generally enforces contracted behavior for a certain set of inputs when a program is being generated (this is called the compilation process). This is meant to ensure the correctness of a program before it's run. Runtime or dynamic checking is used while the program is running. Its purpose is to ensure that the program makes decisions based on the types or subtypes it encounters (runtime polymorphism). You can also combine static and dynamic type checking.

Comparing Clojure with object orientation

Java is a class-based, object-oriented programming language. This object-oriented paradigm was developed in the late 1950s. It grew out of a need to reliably build larger, more complex systems than was previously possible using an imperative programming model (you can read more at `https://en.wikipedia.org/wiki/Imperative_programming`). Object orientation's original computation model focused on data encapsulation and object interaction through message passing versus imperative's model of statements that change the state of a program. These objects can be derived from classes, prototypes, or through some other mechanism. Object-oriented programming was first popularized in languages, such as Simula (you can read more at `https://en.wikipedia.org/wiki/Simula`) and Smalltalk (you can read more at `https://en.wikipedia.org/wiki/Smalltalk`). There are now many major languages that use this programming model, including Java.

It has statically and dynamically typed program elements. This means that you can inspect and dispatch (decide) based on the type of object at runtime. Java variable types are also checked at compile time. Let's take a look at a possible implementation of our `simple-moving-average` function using Java:

```
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Lagging {

   public Stream<Double> readPrices(String path) throws     URISyntaxException , IOException {
       return Files.lines(Paths.get(new URI(path)),         StandardCharsets.UTF_8)
               .map(Double::parseDouble);
    }

    public List listPrices(String path) throws URISyntaxException     , IOException {
        return this.readPrices(path).collect(Collectors.toList());
    }

   public List simpleMovingAverages(String pricesPath, int     tickWindow) throws URISyntaxException , IOException {

        List priceList = this.listPrices(pricesPath);
        List averageList = new ArrayList();

        int begin = 1;
        int end = tickWindow;
        while(end <= priceList.size()) {

            List currentWindow = priceList.subList(begin, end);
            Iterator iter = currentWindow.iterator();
            double total = 0d;
            while (iter.hasNext()) {
                total += (Double)iter.next();
            }

            averageList.add(total / tickWindow);

            begin++;
            end++;
        }
```

```
            return averageList;
        }

        public static void main(String[] args) throws     URISyntaxException , IOException {

            String pricesPath = "file:///Users/timothyw/Projects/
    LaggingJava/src/main/resources/prices.txt";
            int tickWindow = 20;
            Lagging lagging = new Lagging();

            List simpleMovingAverages =        lagging.simpleMovingAverages(pricesPath, tickWindow);
            System.out.println(simpleMovingAverages);
        }
    }
```

The `main` method is the start point of the program. However, before we study it, notice that it exists inside the `Lagging` class. All program state and behavior must be created inside a class or similar structure (interface, abstract class, and so on). The idea is that the class is a template from which we create an actual thing called an object.

Static program elements are those that exist in the class or template itself, not its objects. This is the condition of the `main` method. After compiling this code, there will be a compiled `Lagging` representation that we can run from our computer. This is done by invoking the `java` runner (or runtime environment) and passing in the compiled `Lagging` representation. By default, the `java` runtime invokes the static main method on the class it is given. In this case, when run, `main` creates a string variable that represents the input file on a computer's hard drive. The `Lagging lagging = new Lagging();` expression then creates a new `Lagging` instance or an object. The next expression, `List simpleMovingAverages = lagging.simpleMovingAverages(pricesPath, tickWindow);`, calls the `simpleMovingAverages` method on this object. The result is assigned to another object of the abstract `List`, type which is then printed to the console.

Just by studying the main method and its context, we see that classes and class instantiation is central to where Java places state and behavior. Diving into the `simpleMovingAverages` method, the first expression calls the `listPrices` method on the current object or the `this` reference. `listPrices` then calls out again to the `readPrices` helper method to pull in the `pricelist.txt` data file, divides it into a list of text lines, and then converts each element of this list into a numeric value of the `java.lang.Double` type.

The second expression, `List averageList = new ArrayList();`, simply creates a list that will be added to or mutated later on in the method. The next two expressions set up the `begin` and `end` states to be used to control the `while` loop. `while` loops are control structures in imperative and object-oriented languages that repeatedly execute code inside a block until some condition is met. In our case, we have two while loops. The first `while(end <= priceList.size())` condition will execute everything inside its block, while the `end` variable is below or equal to the length (or size) of the price list. What this block is doing is taking a subset or window of the entire list and calculating the average for this window. The second while loop walks or iterates over the subset or `currentWindow`, using the Iterator type in order to sum all the numbers in this subset. Each time the outer `while` loop is executed, the average is calculated, the result placed into `averageList`, and the `begin` and `end` markers each increment forward until `end` hits the length of the list. When we run this class, the console printed `averageList`, has the average values we expect.

```
[6.971467579790411, 7.464540409139185, 8.010389878709972,
8.595550217357273, 9.199880686839014, 9.803134294595337,
10.391281772750983, 10.960810205584073, 11.519789847777256,
12.085410173832697, 12.657204984470962, 13.512979176946038,
14.336798333665858, 15.033597887711505, 15.93293648884989,
17.236687274535527, 18.017166796811743, 18.747178634896592,
19.36891681775512, 19.88107990363371, 20.352063482865127,
20.809990525663345, 21.292779590969797, 21.65720104327843,
21.88759572553658, 22.0960538285178, 22.522518975747712,
22.921023864595572, 23.300273299533917, 23.677893061861866,
23.791812046695554, 23.92261161858174, 24.13540925343312,
24.28043376993932, 23.99307801388237, 23.69529121331615,
23.444825362117815, 23.313832222382498, 23.30398859426755,
23.34531490845263, 23.386641222637703, 23.52694480589677,
23.67574694970999, 23.906119057816472, 24.25121614021084,
24.211544668391998, 24.23599987692504, 24.305644963968547,
24.243653669023793, 24.31898689789535, 24.369948081810513,
24.364340164808066, 24.518113870232213, 24.573545352458943,
24.63940787919494, 24.75959935515974, 24.806090418384898,
24.772532158349563, 24.774246735296167, 24.857257750991796,
24.765966922838224, 24.67207478924459, 24.61362131993375,
24.52537925344091, 24.58801417608733, 24.586522418381865,
24.532091262456568, 24.56313935471497, 24.393672293029404,
24.188468574045014, 24.05833078712983, 23.919361348321903,
23.947386025442054, 23.872246343092222, 23.637285378280023,
23.515604315240033, 23.594912277922973, 23.649875136387475,
23.505921609914147, 23.30230219819553, 23.255901647048383]
```

However, the way in which we've calculated this information, by manually sliding a current window through the length of the price list, is different from our original Clojure version. The Java version forces us to not just declare types for each program element. It's also a more manual process of managing how lists are processed and all the intermediate types and steps required in this traversal. Clojure's `partition` function lazily gave us a subset of an entire list. `reduce` walks over this list of lists to calculate the average value of each one. Nowhere did we manually manage the order of operations to walk a list. We simply gave calculation instructions at each step. This is what's meant by a more *declarative* style of programming (you can read more at `https://en.wikipedia.org/wiki/Declarative_programming`).

Comparing Clojure with FP and strong typing

Haskell is a statically typed programming language. The type system is a central part of the language and applies to all program elements, even at the most granular level. We'll take a look at how to implement the simple-moving-average function using this system:

```
module Main where


import Data.List.Split(chop)

type Prices = [Float]
type AveragePrices = [Float]


loadPrices :: FilePath -> IO Prices
loadPrices path = do
  priceContents <- readFile path
  let priceLines = (lines priceContents)
  return (map (\x -> read x :: Float) priceLines)


divvy :: Int -> Int -> [a] -> [[a]]
divvy n m lst =
  filter (\ws -> (n == length ws)) choppedl
    where choppedl = chop (\xs -> (take n xs , drop m xs)) lst


average :: [Float] -> Float
average xs = sum xs / (fromIntegral $ length xs)


simpleMovingAverage :: Prices -> AveragePrices
simpleMovingAverage priceList =
  map average divvyedPrices
    where divvyedPrices = divvy 20 1 priceList


main :: IO ()
main = do
  priceList <- loadPrices "prices.txt"
  let priceAverages = simpleMovingAverage priceList
  print priceAverages
```

Let's start at the bottom of the file, the `main` function. This is the entry point of a program. The `main :: IO ()` expression names the function. The part to the right-hand side of `::` declares that the function will return an effective I/O operation, which returns a `()` **Unit** type. A Unit type is akin to a `void` type in Java, which simply means that the function doesn't return anything. By introducing `IO` and its effects, we have to understand that Haskell has a strict policy of containing any operation that doesn't return the value of its evaluation. These are known as impure operations, or side effects.

Pure functional languages are those that do not have any side effects. The computation model is composed entirely of program elements whose operations can be guaranteed at compile time. The goal is to make more provable systems. The result of this is that code tends to be more declarative, as we can see in the preceding Haskell example.

Most Haskell functions return the result of one nested evaluation block. The `main` function evaluation occurs in a `do block` that lets us evaluate several expressions one after the other. `loadPrices` pulls in a large text of price data. Loading data from a filesystem is an operation with side-effects because data is being loaded from outside the system's runtime environment:

```
5.776354340433784
4.819385095126501
4.024983019350133
3.6118300426230148
3.69365835786349
4.247942377070631
5.1220995447382265
6.0754930574679324
```

We then bind that result to `priceList`, which is then used in our `simpleMovingAverage` function, the result of which is printed to the console (the `IO ()` operation). `loadPrices` then divides this data into lines and maps a function over this list to convert each text line into a numeric value (`Float`). Something to note about `loadPrices` is that it explicitly declares the type of data you can input, and the type of data that will be returned. `loadPrices :: FilePath -> IO Prices` declare the function, and `FilePath` and `Prices` are types that we've declared in the previous section. Both types are lists of `float` values. The `IO` qualifier again declares an effectful operation that will be returned, only this time on `Prices` (a list of `Float` values).

Simply by looking at the `main` and `loadPrices` functions, we already have an idea of Haskell's general computation model. The type system is pervasive and is used all the way down to the most granular program elements. This is a tool meant to maintain pure operations that are free of side effects. Any required side effects are contained to effectful types and explicitly declared when using these types. The result is more compile-time guarantees of program correctness, and peripherally, a more declarative code style. Clojure is not a pure functional language because it allows side effects where they are pragmatically needed. It is also more loosely typed as program element types do not have to be declared and arranged at compile time. Clojure veers toward fewer types that are consistent and can represent a wider array of data. This also means more functions and more general functions can operate on this data. While Clojure is more declarative than Java, it emphasizes **List processing** (how we get the name **Lisp**). You can read more about Lisp at `https://en.wikipedia.org/wiki/Lisp_(programming_language)`. Lisp's macro system, or the ability to rewrite code on the fly, came out of its data-centric and homoiconic property. It's also what enables Lisps and Clojure, in this case, to have several third-party options for static analysis. `core.contracts` (you can read more at `https://github.com/clojure/core.contracts`) and `core.typed` (you can read more at `https://github.com/clojure/core.typed`), for example, are just two third-party tools that offer static code analysis using the **design by contract** and **gradual typing** approaches, respectively.

Haskell's more declarative code style is evident in the `simpleMovingAverage` function. The `simpleMovingAverage :: Prices -> AveragePrices` declaration tells us that it takes the `Prices` type (a list of `Float` values) and returns the `AveragePrices` type (also a list of `Float` values). `map` acts the same as it does in Clojure, mapping the `average` function over the `divvyedPrices` list. The `divvyedPrices` list is created from the original `priceList` using the `divvy` function we've created. Our `divvy` function behaves in the same way as the Clojure `partition` function, returning a list of `n` items, where the start of each partition is separated by `m` elements. The `average` function simply takes a list of `Float` values and divides their sum by their size.

The remaining `import` function simply pulls in the `chop` function from the `Data.List.Split` module, and the `module` declaration is Haskell's way of grouping program elements. These include functions, data, type definitions, and so on. It's Haskell's namespacing facility and has the same purpose as Clojure's `ns` macro. I'm using a Haskell build tool called `cabal` (you can read more at `https://www.haskell.org/cabal`). If we compile our Haskell file, we'll get an executable that runs, using the the `prices.txt` file, as input:

```
\\ cabal build
Building lagging-haskell-0.1.0.0...
Preprocessing executable 'lagging-haskell' for lagging-haskell-0.1.0.0...
[1 of 1] Compiling Main             ( Main.hs, dist/build/lagging-
haskell/lagging-haskell-tmp/Main.o )
Linking dist/build/lagging-haskell/lagging-haskell ...


\\ cabal run
Preprocessing executable 'lagging-haskell' for lagging-haskell-0.1.0.0...
Running lagging-haskell...
[7.260286,7.70551,8.211639,8.776141,9.384563,10.015532,10.6473875,11.2645
855,11.862074,12.446443,13.012064,13.83844,14.66226,15.385584,16.131344,1
7.447102,18.75085,19.480864,20.09607,20.608353,21.086105,21.557089,22.058
533,22.446215,22.703249,22.940306,23.395824,23.822289,24.226925,24.604544
,24.973045,25.071888,25.18419,25.378178,25.50724,25.209455,24.908524,24.6
62724,24.543427,24.55034,24.591665,24.775486,24.829182,24.952164,25.30392
6,25.511312,25.535767,25.611546,25.547924,25.61414,25.650028,25.625921,25
.760883,25.800354,25.855785,25.97283,26.023987,26.002125,26.020596,26.103
607,26.15481,25.97431,25.890036,25.923187,25.84811,25.910746,25.907635,25
.80542,25.764154,25.519506,25.314304,25.315907,25.229626,25.154488,24.970
709,24.779993,24.790949,24.897942,24.835285,24.599857,24.464247]
```

Summary

Hopefully, you've taken away a broader scope of the available Clojure functions, and how to use them to access, update, and compose data structures. This applies to scalars, collection types, and composite data types. You should also have a good grasp of I/O operations, which are Clojure's options when dealing with concurrency and parallelism and an understanding of the difference between the two.

We also stepped back and compared our Clojure `simple-moving-average` function to equivalent versions in an object-oriented language (Java) and a purely functional, statically typed language (Haskell). This is all meant to orient you to Clojure's approach to computation and the different trade-offs that it makes.

The next chapter will consider our existing data structures output from the `generate-prices`, `simple-moving-average`, `exponential-moving-average`, and `bollinger-band` functions, and store each tick progression for later look ups. We'll also deal with `IO` and its side effects and the best way to contain them.

7
Dealing with Side Effects

We're going to try and take the data we've generated so far and save it so we can look it up later. In functional programming, reading and writing from the outside world is called a side effect because such an operation does not directly deal with inputs or outputs that are passed into or returned from a function. This goes back to the idea of a pure functional language that's closer to a mathematical expression. The following properties that result from this are more practical in nature:

- Given the same input, a function will always return the same output.
- If data from more than one pure function don't depend on each other, their orders can be reversed or they can all be run in parallel.
- There are other benefits, such as the ability to remove expressions or change evaluation strategies. However, this is usually the domain of the compiler (the thing that translates your code to 0s and 1s or some executable form).

So, we'll deal with IO, its side effects, and the best way to contain them. In this chapter, we'll cover these topics:

- Writing out data
- The Extensible Data Notation (EDN)
- Devising a persistence strategy
- Consuming from a data stream
- Using a componentized architecture

Simple writing

Things, such as threads, input, or output streams to files consume your computer's resources. Thus, it's important to manage them efficiently. The `with-open` macro takes any open readers and writers, lets you perform your evaluation that reads or writes to them, and then closes these resources on completion. This is a good start to ensuring that we're making efficient use of our resources.

The simplest way to try this is to write some data to your filesystem. Open your REPL and evaluate the following code. The `(io/writer "datafile")` expression creates a Java writer (`java.io.BufferedWriter`) object, which we then use to write the data. The `(.write wr (str '(:some :data)))` method is another example of Clojure's interoperability with Java, that is, we call the `.write` method on the `wr` Java object, all of which are in Clojure. The directory in which you started your REPL should now have a file called `datafile` with the following contents `(:some :data)`:

```
(require '[clojure.java.io :as io])
(with-open [wr (io/writer "datafile")]
  (.write wr (str '(:some :data))))
```

If we run it again and submit some new data, the file's old contents should be gone and replaced by what you've just written out:

```
(with-open [wr (io/writer "datafile")]
  (.write wr (str '(:new :data))))
```

It turns out that the `spit` function uses `with-open` to write the content of your data to a file. This is a bit simpler to use, so we'll go with it:

```
(spit "datafile" '(:foo :bar))
```

Extensible Data Notation

Notice that, so far, we've just written out lists of keywords. The output file shows the exact data we submitted in our REPL. Recall that Clojure, like all Lisps, is homoiconic. This means that your code is also data. By comparison, you cannot simply write out Java or Haskell program elements. You can write out their code's string equivalents. However, you can't do this with literal code as it lives in the runtime. Clojure has used this homoiconic quality to distill a subset of itself into a data format called the **Extensible Data Notation** (**EDN**) (you can read more at `https://github.com/edn-format/edn`).

EDN data values should be considered as immutable, having no notion of any program reference in a language's runtime (that is no notion of object identity). As such, this data format can be used to save data, transfer data between programs that are in different languages, and so on. EDN has its own built-in elements, such as integers, strings, lists, keywords, and so on.

There is also a facility to extend notations with tags such as (`#uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"`). Tagged elements describe custom data types and allow you to extend the notation with your own types (hence the *extensible* moniker). EDN has its own built-in tagged types. This will be important later on when we want to write out the time or instance of a particular stock price (`#inst "1985-04-12T23:20:50.52Z"`).

So, we can rightly spit out our data to a file using the `.edn` file extension `(spit "datafile.edn" '(:more :stuff) :append true)`:

```
(defn generate-file-name [fname]
  (str "data/" fname))

(defn write-data [fname data]
  (let [full-path (generate-file-name fname)]

    (io/make-parents full-path)
    (spit full-path (list (apply pr-str data))
           :encoding "UTF-8")))

(write-data "datafile.edn" '(:one :two :three))
```

Devising a persistence strategy

Considering the structure of data and how it will be used later, may inform us about how we write it now. We know that we have an infinite sequence of data that we need to write. In the software profession, these are known as data streams. They can be in the form of video feeds from CCTV cameras or social media content, or in our case, a continuously updating stock price. This means that we will never be able to collect all the data in memory and then write it out. Knowing this, we need to devise a strategy to collect and save our data in a manageable way.

In fact, input and output streams are conceptual ways of handling an infinite stream of data. Clojure leverages Java's streaming API in the `clojure.java.io` namespace (you can read more at `http://clojuredocs.org/clojure.java.io`). We're going to itemize the operations we need to perform in order to decide whether to use an approach that directly spits out the contents of some data versus incrementally streaming out this data.

To begin, we need to store each tick progression incrementally so that it can be looked up later on. These are the operations and data components that are relevant:

- Storing a raw tick list
- Storing a simple moving average
- Storing a exponential moving average
- Storing a bollinger band
- Relating them by time

Streaming new data to an existing file nominally means appending to the end of this file. For example, evaluating `(spit "datafile.edn" '(:more :stuff) :append true)` will append to the end of the previous file, producing the following content `(:foo :bar)(:more :stuff)`. The format, as we've just seen, won't work for our purposes. We'd want to see something, such as `(:foo :bar :more :stuff)`, which would involve the following:

- Opening up an existing file
- Reading its content
- Appending to the content
- Writing out the content again

This is not an efficient use of our computer's resources, given that this update may take place every second or whenever a stock price's data gets updated.

Given the frequency of the data updates, it makes more sense to collect a set of data and write this batch out to a file. Deciding the threshold at which to write out a batch of data should entail its own analysis. This would involve knowing how often a data update would take place, the size of our program's memory, where this data will be used later, and so on. For the sake of simplicity, however, I'll randomly pick a list size of 320 ticks at which we can flush out the content of our data. I'll also separately write out each data analytic (raw tick list, simple moving average, and so on). This separation is possible because the analytics can later be related to time (the EDN `#inst` tag). In this way, we can arrive at the following features:

- Save data as separate size-manageable files
- Each analytic is easily distinguished without any extra labeling; this will be necessary if they were comingled in the same file

These features will also be useful when later looking up our data. We should be able to do the following:

- Look up a price based on a specific period of time
- Look up a price based on a time range
- Look up a data point based on a specific price
- Look up a data point based on a price range

With our data neatly arranged, we should be able to layer additional analytics on top of it. For example, by looking at the composition of our source tick list, simple moving average, exponential moving average, and bollinger band, we can start to derive some buy and sell signals.

Consuming from a data stream

In order to consume from an infinite stream of data, let's review the approaches discussed in *Chapter 6*, *Surveying the Landscape*.

Futures and promises only offer a one-time asynchronous task and are not suitable in this case. Also, we need some kind of list into which a data producer can insert ticks and analytics data. A consumer of this data would then be able to asynchronously take from the list. This is what a queue provides conceptually.

The `clojure.lang.PersistentQueue/EMPTY` option is nonblocking. So, if there's no data in a queue, the consumer would have to continuously ask for data until it arrives (known as **polling**). This is not quite what we want. Ideally, the consumer should wait (or block) until the data is available for use.

Java's persistent queues, paired with threads (or other concurrency tool) would mean having to deal with extra memory resources that threads impose, and Java interoperability issues such as thread monitoring and so on. Reducers, simply offer parallelism. These would be useful later on as an optimization but they're not necessary in the first pass of our design.

Recall that for the concurrency model of Communicating Sequential Processes, Clojure has a third-party library called `core.async`. Clojure's `core.async` library offers queues and concurrency in the form of channels and go blocks. Channels can also be easily passed around, and consumers block until data becomes available. These are good options for us. We'll set up the remainder of the chapter by first adding the `core.async` and `component` libraries to our project:

```
(defproject edgar "0.1.0-SNAPSHOT"
:dependencies [[org.clojure/clojure "1.7.0"]
               [clj-time "0.9.0"]
               [org.clojure/math.numeric-tower "0.0.4"]
               [org.apache.commons/commons-math3 "3.5"]
               [org.clojure/core.async "0.1.346.0-17112a-alpha"]
               [com.stuartsierra/component "0.2.3"]])
```

When done, the `project.clj` file should look similar to the preceding code (`leiningen` will already have added the entries: `description`, :`url`, and `:license` to `defproject`). You can now restart REPL environment.

1. The first order of business is to create an infinite data stream and pull tick data from it in chunks of 320. This is possible with the functionality we've already created. Recall that in *Chapter 5*, *Traversing Data, Branching, and Conditional Dispatch*, our refactored `datasource/generate-prices` function was used to generate raw price data. We can plug the result of this to our `core/generate-timeseries` from *Chapter 2*, *First Principles and a Useful Way to Think*. Now, it's just a matter of taking the first 320 elements and then the remaining part of the list. In your REPL, begin by pulling the necessary namespaces and aliases. Then, perform the subsequent evaluations:

   ```
      (require '[edgar.core :as core]
            '[edgar.analytics :as analytics]
            '[edgar.datasource :as datasource]
            '[clojure.core.async :as async :refer [go go-loop
            chan close! <! >!]]))

   (def price-list (datasource/generate-prices))
   (def time-series (core/generate-timeseries price-list))

   (def prices (take 320 time-series))
   (def remaining (drop 320 time-series))
   ```

2. With our price data, we can call our analytic functions. All the functions are available in *Chapter 4, Strategies for Calculating and Manipulating Data*; the `four/simple-moving-average` function takes this list of 320 ticks and calculates their averages in a window of 20 ticks. This window will slide forward until the end of the list is reached. The price list and results from the `analytics/simple-moving-average` function are then fed into the `analytics/exponential-moving-average` and `analytics/bollinger-band` functions to produce these results:

   ```
   (def sma (analytics analytics/simple-moving-average {} 20 prices))
   (def ema (analytics analytics/exponential-moving-average {} 20
   prices sma))
   (def bol (analytics analytics/bollinger-band 20 prices sma))
   ```

3. It's simple for us to now take all the data we've calculated and save it in a file. We can use the `write-data` function to simply write our data to named files:

   ```
   (write-data "ticks.edn" prices)
   (write-data "sma.edn" sma)
   (write-data "ema.edn" ema)
   (write-data "bol.edn" bol)
   ```

4. Now that we know that there will be a producer and consumer of data, let's use the pulled in `core.async` library. Creating a channel is simple. We'll use the `>!` function to put data into this channel within the context of the `go` block. This block is the concurrency mechanism that `core.async` provides, which manages threads and concurrency in the background. After those four pieces of data are on the channel, we can pull them off with the `<!` function, again within the context of the `go` block. Try evaluating this in your REPL:

   ```
   (let [c (chan)]

     ;; 4. put generate prices into core.async channel
     (go (>! c {:ticks prices}))
     (go (>! c {:sma sma}))
     (go (>! c {:ema ema}))
     (go (>! c {:bol bol}))

     ;; 5. analytics reads data from channel
     (println (<!! (go (<! c))))
     (println (<!! (go (<! c))))
     (println (<!! (go (<! c))))
     (println (<!! (go (<! c))))

     (close! c))
   ```

Using a componentized architecture

Now that we've laid out the preceding steps, we'll put the functionalities into a more cohesive package. A queue of data is runtime state that has to be managed. Recall the component architecture that's widely used in the Clojure community. This architecture is the result of an academic paper called *Out Of The Tarpit* in which the authors have argued that complexity is the biggest nemesis to working software (you can read more at `http://shaffner.us/cs/papers/tarpit.pdf`). Any incidental complexity should be avoided or removed, and things, such as runtime state, should be cordoned off to a small managed component of the program. This is important because it reflects a componentized architecture that many banks use. It allows for an incremental approach to building out and architecting core banking systems. Many industry participants adopt what's known as a **Service-Oriented Architecture** (**SOA**) as part of a suite of best practices. There's a core banking platform or environment that puts basic core banking capabilities in place quickly. Then, gradually, a bank can add components sourced from many different vendors. This is done in a way that ensures open standards and keeps the bank flexible enough to make future investments. The benefits of this approach are such that the bank gets to choose the best tools for its needs. These are tools and technologies that let management ask key questions pertaining to assets, liability, liquidity, risk, and so on, in a timely manner (you can read more at `https://en.wikipedia.org/wiki/Core_banking` and `http://www-05.ibm.com/cz/businesstalks/pdf/Core_Banking_Modernization_Point_of_View.PDF`).

Similarly, a component isolates and manages the life cycles of units of code that have a runtime state. So, we can be confident in using this approach. So far, we've entered code into an REPL. Now, you can create a file called `src/edgar/component.clj`. At the top of our namespace declaration, we're going to require the same libraries as those in the preceding section. However, we're also going to add a component library and some date formatting libraries:

```
(ns edgar.component
  (:require [clojure.java.io :as io]
            [clojure.core.async :as async :refer [go go-loop chan
            close! <! >!]]
            [com.stuartsierra.component :as component]
            [clj-time.format :as fmt]
            [edgar.core :as core]
            [edgar.analytics :as analytics]
            [edgar.datasource :as datasource])
  (:import [java.text SimpleDateFormat]))
```

Now we can begin to write our code. After reviewing component's documentation, we can put up an initial scaffolding with components for the infinite stream of time series data, and consumers of this data:

```
(defrecord Timeseries []
  component/Lifecycle

  (start [component]
    (let [c (chan)]
      (assoc component :channel c)))

  (stop [component]
    (let [c (:channel component)]
      (close! c)
      (dissoc component :channel))))

(defrecord Consumer [timeseries]
  component/Lifecycle

  (start [component]
    (assoc component :channel (:channel timeseries)))

  (stop [component]
    (dissoc component :channel)))

(defn new-timeseries []
  (map->Timeseries {}))

(defn new-consumer []
  (map->Consumer {}))


(defn build-system []
  (component/system-map
   :tms (new-timeseries)
   :cns (component/using
          (new-consumer)
          {:timeseries :tms})))

(def system (build-system))
```

When we're done, we will be able to start our system with the `component/start` function. Often, when we `def` a symbol, it indicates a simple scalar value or collection. However, when we called `component/system-map` (within `build-system`), it returned a Clojure `var` (one of Clojure's container values that allows controlled updates). So, to change the system `var`, we'll use `alter-var-root` to atomically assign a new value, thereby avoiding concurrent updates. Thus, in a scenario where `def` would redefine an existing `var`, `alter-var-root` creates a new value, which is a succession of the previous value. You can now inspect the system map and see the `core.async` channel that's shared between the time series and consumer components. The `clojure.pprint/pprint` call neatly prints and formats structured data:

```
> (alter-var-root #'system component/start)
> (clojure.pprint/pprint system)
{:tms
 {:channel
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]},
 :cns
 {:timeseries
  {:channel
   #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]},
  :channel
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x14d1e55
"clojure.core.async.impl.channels.ManyToManyChannel@14d1e55"]}}
```

We can now add the `send-data!` and `receive-data` functions that are used by the `time-series` and `consumer` components, respectively:

```
(defn send-data! [channel time-series]

  (go-loop [prices (take 320 time-series)
            remaining (drop 320 time-series)]

    (let [sma (analytics/simple-moving-average {} 20 prices)
          ema (analytics/exponential-moving-average {} 20 prices
          sma)
          bol (analytics/bollinger-band 20 prices sma)]

      (>! channel {:ticks prices :sma sma :ema ema :bol bol})
      (Thread/sleep 1000))

    (recur (take 320 remaining)
          (drop 320 remaining))))

(defn receive-data! [channel]

  (go-loop [data (<! channel)]
           (println data)
           (if-not (nil? data)
             (recur (<! channel)))))

(defrecord Timeseries []
  component/Lifecycle

  (start [component]
    (let [c (chan)
          price-list (datasource/generate-prices)
          time-series (core/generate-timeseries price-list)]

      (send-data! c time-series)
      (assoc component :channel c)))

  (stop [component]
    (let [c (:channel component)]
      (close! c)
      (dissoc component :channel))))

(defrecord Consumer [timeseries]
  component/Lifecycle

  (start [component]
    (let [channel (:channel timeseries)]

      (receive-data! channel)
      (assoc component :channel channel)))

  (stop [component]
    (dissoc component :channel)))
```

The `receive-data!` expression just prints out what it's consumed. Once we see this working successfully, we can improve `receive-data!` to write the data out in the manner we laid out in the *Devising a persistence strategy* section:

```
(defn receive-data! [channel]

  (go-loop [data (<! channel)]

    (if-not (nil? data)
      (do
        (let [{ticks :ticks sma :sma ema :ema bol :bol} data
              timestamp (-> ticks last :last-trade-time .toString)
              generate-file-name-with-timestamp-fn (fn [fname]
              (str timestamp "-" fname))]

          (write-data (generate-file-name-with-timestamp-fn
          "ticks.edn") ticks)
          (write-data (generate-file-name-with-timestamp-fn
          "sma.edn") sma)
          (write-data (generate-file-name-with-timestamp-fn
          "ema.edn") ema)
          (write-data (generate-file-name-with-timestamp-fn
          "bol.edn") bol))
        (recur (<! channel))))))
```

Run your system for a few seconds by calling `(alter-var-root #'system component/start)`. When you think you've collected enough data, stop your system with `(alter-var-root #'system component/stop)`. When you look at your `data/` directory, you should see output files that look something like this:

```
$ ls data/
.                                   2015-08-09T23:10:43.979Z-ticks.edn
2015-08-09T23:34:30.979Z-bol.edn    2015-08-09T23:49:21.979Z-ema.edn
2015-08-10T00:05:30.979Z-sma.edn    2015-08-10T00:22:00.979Z-ticks.edn
2015-08-10T00:45:34.979Z-bol.edn

..                                  2015-08-09T23:18:25.979Z-bol.
edn    2015-08-09T23:34:30.979Z-ema.edn    2015-08-09T23:49:21.979Z-sma.
edn    2015-08-10T00:05:30.979Z-ticks.edn  2015-08-10T00:29:57.979Z-bol.
edn    2015-08-10T00:45:34.979Z-ema.edn

2015-08-09T23:02:58.979Z-bol.edn    2015-08-09T23:18:25.979Z-ema.edn
2015-08-09T23:34:30.979Z-sma.edn    2015-08-09T23:49:21.979Z-ticks.edn
2015-08-10T00:13:46.979Z-bol.edn    2015-08-10T00:29:57.979Z-ema.edn
2015-08-10T00:45:34.979Z-sma.edn
```

```
2015-08-09T23:02:58.979Z-ema.edn    2015-08-09T23:18:25.979Z-sma.
edn    2015-08-09T23:34:30.979Z-ticks.edn  2015-08-09T23:57:20.979Z-bol.
edn    2015-08-10T00:13:46.979Z-ema.edn    2015-08-10T00:29:57.979Z-sma.
edn    2015-08-10T00:45:34.979Z-ticks.edn

2015-08-09T23:02:58.979Z-sma.edn    2015-08-09T23:18:25.979Z-ticks.edn
2015-08-09T23:41:43.979Z-bol.edn    2015-08-09T23:57:20.979Z-ema.edn
2015-08-10T00:13:46.979Z-sma.edn    2015-08-10T00:29:57.979Z-ticks.edn

2015-08-09T23:02:58.979Z-ticks.edn  2015-08-09T23:26:10.979Z-bol.edn
2015-08-09T23:41:43.979Z-ema.edn    2015-08-09T23:57:20.979Z-sma.edn
2015-08-10T00:13:46.979Z-ticks.edn  2015-08-10T00:37:28.979Z-bol.edn

2015-08-09T23:10:43.979Z-bol.edn    2015-08-09T23:26:10.979Z-ema.edn
2015-08-09T23:41:43.979Z-sma.edn    2015-08-09T23:57:20.979Z-ticks.edn
2015-08-10T00:22:00.979Z-bol.edn    2015-08-10T00:37:28.979Z-ema.edn

2015-08-09T23:10:43.979Z-ema.edn    2015-08-09T23:26:10.979Z-sma.edn
2015-08-09T23:41:43.979Z-ticks.edn  2015-08-10T00:05:30.979Z-bol.edn
2015-08-10T00:22:00.979Z-ema.edn    2015-08-10T00:37:28.979Z-sma.edn

2015-08-09T23:10:43.979Z-sma.edn    2015-08-09T23:26:10.979Z-ticks.edn
2015-08-09T23:49:21.979Z-bol.edn    2015-08-10T00:05:30.979Z-ema.edn
2015-08-10T00:22:00.979Z-sma.edn    2015-08-10T00:37:28.979Z-ticks.edn
```

Summary

We successfully used a componentized architecture to encapsulate all the functionality that we developed so far. We were also able to develop a persistence strategy to write out our core tick list and accompanying analytics in the EDN format. This was all done by writes to functions that employed the `with-open` macro, which contained the IO that had side effects and cleaned up resources when finished. The next chapter will explore ways of searching for data we've saved.

8
Strategies for Using Macros

The previous chapter saw us saving the data we read and analyzed. This gave us the chance to develop a persistence strategy which writes our data in the EDN format and uses a component pattern. Let's now turn our attention to reading this data and seeing how far we can go while searching for it. In this chapter, we'll roughly cover the following topics:

- Simple reading
- An example of a regular expression
- Querying and looking up data
- A simple query language
- Variable argument functions
- The `:pre` and `:post` function conditions
- The `juxt` higher order function
- Separating the OR/AND lookups
- Deriving a small query language using macros

Simple reading

Reading in EDN (or any other) files follows the same pattern as writing. This is because our writes employ the `with-open` macro, which may contain any IO that has side effects, and cleans up resources when finished. The `slurp` data is the mirror image of `spit`, in that it reads in data from your filesystem using the `with-open` macro.

To try it out, open up your REPL and add the following code. Replace the `08-15-2015-04:55:56-ticks.edn` data with any EDN data file that you've written out from the previous chapter. The `slurp` command will simply read in any data that is in this file. In this case, there will be a bunch of characters that make up an EDN data structure. We can then transform these characters into Clojure data structures by simply using the built-in `edn/read-string` function from the `clojure.edn` namespace:

```
(require '[clojure.edn :as edn])

;; read one file usung slurp
(def one-file (slurp "data/08-15-2015-04:55:56-ticks.edn"))
(def one-edn (edn/read-string one-file))
```

The preceding code demonstrates how easy it is to read just one file. However, this is generally a good approach for small to medium-sized files. Larger files will warrant reading directly from a stream. But we won't cover that option here. What we do want to cover is how to recursively read all files under a directory. This is easily done using the `file-seq` function. Simply give the function the directory from which you want to read and it will suck up all the files under it. Be sure to give it a Java type of `java.util.File`. This is easily done with the `file` function in the `clojure.java.io` namespace.

When you refer to the following code, `many-files` will just be a sequence of `java.util.File` (including the source directory), which has been picked up. We can try this out by taking one of the files and slurping it. As shown in our simple file read example, we have all the characters present in this file. Since it is an EDN document, we can put the stream of characters (a string) into the EDN namespace's `read-string` function and get back the Clojure data structure that was represented in the file:

```
(require '[clojure.java.io :as io])
(def many-files (file-seq (io/file "data/")))

;; slurp takes a file name or a file object
(def second-file (slurp (second many-files)))

;; we've now demonstrated round tripping of our data
(def second-edn (edn/read-string second-file))
```

Functions for querying a system

If you've followed the progress from the last chapter, per data chunk, you would have written out four kinds of files that represent all the data we've generated:

```
<file-name>-tick.edn
<file-name>-sma.edn
<file-name>-ema.edn
<file-name>-bol.edn
```

Since we've named the files appropriately in our list of `many-files`, we can use the `filter` function to isolate on any condition—in this case, we'll isolate only one type of file.

An example of a regular expression

For our predicate function, we can use a regular expression to search for a string pattern in a filename. Clojure represents regular expressions with a string prefixed by a hash such as `#""`. So, the `re-matches` function, for example, takes a regular expression and string and returns the first match it finds, if any:

```
(re-matches #"x" "foobar")
;; nil

(re-matches #"foo.*" "foobar")
;; "foobar"
```

The returned value (or the absence of one, which is `nil`) is considered truthy, which just means that we can use it in an `if` condition or a predicate. So, our filter function will look something like this:

```
;; filter on the type of file
(filter #(re-matches #".*\-ticks.edn" (.getName %))
        many-files)
```

Here, we filter over our many-files result list. The predicate is an anonymous function that checks a condition by calling `re-matches` on each element in the list using the `#".*\-ticks.edn"` name pattern. Each element happens to be a Java `java.io.File` class with the `getName` method. Therefore, any filename that matches our name pattern will satisfy the filter.

A basic lookup

Our `re-matches` expression is just one kind of a predicate that filters on a type of file. We can also filter the content in files. This time, however, let's pick a particular time instance that you've saved. My filesystem has `#inst 2015-08-15T17:18:51.352-00:00`. With regard to the following code, we can understand it by first looking at the outer `map` function, which itself takes a mapping function, and a list of files. This input list is filtered due to the constraint of being a file—no directories are allowed. Therefore, it makes sense to look at the inner `filter` function. It takes a (`pred-fn`) predicate function and inputs EDN data, for example, (`input-edn`). The predicate function qualifies that the `:last-trade-time` value in each map is exactly the time instance of `#inst "2015-08-15T17:18:51.352-00:00"`. However, `input-edn` is a little more interesting. Each file that's passed to the mapping function calls our created `read-fn`. The `read-fn` function is another example of how the `comp` function is used. Recall that `comp` simply creates a function by composing other functions together. So, in this case, `read-fn` is a composition of `edn/read-string` and `slurp`. When used, the input to `read-fn` will first go to `slurp`, the results of which are then chained to `edn/read-string`:

```
(map (fn [each-file]
       (let [read-fn (comp edn/read-string slurp)    ;; -> use of comp function
             pred-fn #(= #inst "2015-08-15T17:18:51.352-00:00" (:last-trade-time %))
             inp-edn (read-fn each-file)]

         (filter pred-fn
                 inp-edn)))

     (filter #(.isFile %) many-files))


'(({:last-trade-time #inst "2015-08-15T17:18:51.352-00:00",
    :last-trade-price 101.90402553222992}
   {:last-trade-time #inst "2015-08-15T17:18:51.352-00:00",
    :last-trade-price 101.63143059175933})
  ()
  ()
  ()
  ()
  ()
  ()
  ()
  ()
  ()
  ()
```

```
    ()
    ()
    ()
    ()
    ()
    ()
    ()
    ()
    ())
```

Running this expression on all my files shows that only two maps have this instant. This is to be expected since the time constraint is so specific. However, `map` will return an empty list for each element where (`filter pred-fn inp-edn`) doesn't match and return anything. So, the output format is not easy to look at.

Flattening structures

We can clean up the preceding nested expression by simply flattening the result list. This can again easily be done with the `flatten` function. It takes a nested combination of sequential data structures (such as lists, vectors, and so on) and gives back content in a single flat sequence, removing empty sequences and maintaining the shape of associative structures:

```
(flatten '({:a {:b {:c 1}}}
           {:g :h}
          ({:e :R})))

;; ({:a {:b {:c 1}}} {:g :h} {:e :R})


(flatten '({:a {:b {:c 1}}}
          ({:g :h}
          ({:e :R}))))
;; ({:a {:b {:c 1}}} {:g :h} {:e :R})
```

The following code is the same as the expression we encountered earlier, but with its results flattened and wrapped in a function definition. In your REPL, define this function, replacing the string to the right-hand side of `#inst` along with a date format that was written to your filesystem. After creating the function, simply evaluate it by calling `(lookupfn)`:

```
(defn lookupfn []

  (flatten
   (map (fn [x]
           (let [read-fn (comp edn/read-string slurp)
           pred-fn #(= #inst "2015-08-15T17:18:51.352-00:00"                (:last-trade-time %))
                 inp-edn (read-fn x)]

             (filter pred-fn
                     inp-edn)))

        (filter #(.isFile %) many-files))))

(lookupfn)

;; ({:last-trade-time #inst "2015-08-15T17:18:51.352-00:00", :last-
trade-price 101.90402553222992} {:last-trade-time #inst "2015-08-
15T17:18:51.352-00:00", :last-trade-price 101.63143059175933})
```

A more expressive lookup

Let's clean up our code by factoring out the filtered list and predicate function. This has the additional benefit of parameterizing predicates to the function and the file list in which we want to search. Create a file called `src/edgar/readdata.clj` and add this namespace with the `require` definition:

```
(ns edgar.readdata
  (:require [clojure.java.io :as io]
            [clojure.edn :as edn]))
```

Start to author the file by adding the following code:

```
(defn lookupfn [flist pred-fn]

    (flatten
     (map (fn [x]
            (let [read-fn (comp edn/read-string slurp)
                  inp-edn (read-fn x)]
              (filter pred-fn
                      inp-edn)))
          flist)))
```

Now we can search for data using a range of criteria. With regard to the data we've written, we can look up tick data based on the following criteria:

- A specific time
- After a specific time
- Before a specific time

- A time range
- A specific price
- Above a specific price
- Below a specific price
- Price range

These predicates encode the criteria we've just listed:

```
(defn specific-time-pred [inst]   ;; -> functions returning functions
  #(= inst (:last-trade-time %)))

(defn time-after-pred [time]
  #(.after (:last-trade-time %) time))

(defn time-before-pred [time]
  #(.before (:last-trade-time %) time))

(defn time-range-pred [lower upper]
  #(and (.after (:last-trade-time %) lower)
        (.before (:last-trade-time %) upper)))

(defn specific-price-pred [price]
  #(= price (:last-trade-price %)))

(defn price-abouve-pred [price]
  #(> (:last-trade-price %) price))

(defn price-below-pred [price]
  #(< (:last-trade-price %) price))

(defn price-range-pred [lower upper]
  #(and (> (:last-trade-price %) lower)
        (< (:last-trade-price %) upper)))
```

We can also try out a few of these predicates on the lookup function we just created. Try these out on your system using the time instants that were written to your disk:

```
(def files (filter #(.isFile %) many-files))


(lookupfn files (specific-time-pred #inst "2015-08-15T17:18:51.352-
00:00"))
```

```
(lookupfn files (time-range-pred #inst "2015-08-15T17:18:00.000-00:00"
#inst "2015-08-15T17:19:00.000-00:00"))

(lookupfn files (specific-price-pred 4.028309189176084))

(lookupfn files (price-range-pred 6 10))
```

A simple query language

So far, we have a very rudimentary lookup function that takes one predicate at a time. It would be nice to have a way to pass many conditions that a lookup function would satisfy so that multiple expressions are possible, as follows:

```
(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00")
(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00"
        :time-before #inst "2015-08-15T17:19:00.000-00:00")

(lookup :price-abouve 12 :price-below 20)

(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00"
        :time-before #inst "2015-08-15T17:19:00.000-00:00"
        :price-abouve 12
        :price-below 20)
```

Variable argument functions

To implement that syntax, the first thing our function should do is accept a variable list of arguments. Clojure functions let us do this by inserting an ampersand before a symbol that will contain our arguments. Try copying the following simple function into your REPL:

```
(defn foobarfn [& manyargs-ofanyname]
  (println manyargs-ofanyname)
  (println (type manyargs-ofanyname)))

(foobarfn 1 2 3 4 4 5 6 7)
;; (1 2 3 4 5 6 7)
;; clojure.lang.ArraySeq

(foobarfn :one :two :three)
;; (:one :two :three)
;; clojure.lang.ArraySeq

(foobarfn '("a" "s" "d" "f"))
;; (("a" "s" "d" "f"))
;; clojure.lang.ArraySeq
```

We can now call this function with any number of arguments. Passing seven integers to the function is just as valid as passing three keywords or one list.

The :pre and :post function conditions

The other thing that our function will want to do up front is ensure that parameters are submitted in pairs. Again, Clojure provides a facility to assert conditions with the `:pre` and `:post` assertions. These let you provide conditions that must be satisfied when entering or exiting your function. Simply put a map with the `:pre` and `:post` keys at the very beginning of your function in order to do this. The values should be vectors of conditions that you want to be true in both cases:

```
(defn onefn [one]
  {:pre [(not (nil? one))]
   :post [(pos? %)]}

  one)

(onefn nil)
;; java.lang.AssertionError: Assert failed: (not (nil? one))

(onefn -1)
;; java.lang.AssertionError: Assert failed: (pos? %)

(onefn 5)
;; 5
```

Try entering the preceding function in your REPL. Calling it with parameters that fall outside the `:pre` and `:post` constraints will throw an `AssertionError`. Keep in mind that you can put as many assertions as you like in either vector of the conditions:

```
(defn lookup [& constraints]
  {:pre [(even? (count constraints))]}

)
```

We can now use these two features to begin our function. Let's use variable arguments and ensure that these constraints are in pairs. Take a look at the following function and try to glean what it's doing:

```
(defn load-directory [fname]
  (filter #(.isFile %) (file-seq (io/file fname))))

(defn lookup [& constraints]
  ;; ensure constraints are in pairs -> Preconditions
  {:pre [(even? (count constraints))]}

  ;; map over pairs - find predicate fn based on keyword - partially apply fn with arg
  (let [files (if (some #{:source} constraints)

              (let [source-source (comp (partial filter #(= :source (first %1)))
                                        (partial partition 2))
                      source-value (comp second source-source)
                      source-key (comp first source-source)]

                  (if (string? source-key)
                    (load-directory (source-key constraints))
                    source-value))

                (load-directory "data/"))

        constraint-pairs (->> constraints
                              (partition 2)
                              (remove #(= :source (first %))))

        find-lookup-fn (fn [inp]
                         (case inp
                           :time specific-time-pred
                           :time-after time-after-pred
                           :time-before time-before-pred
                           :price specific-price-pred
                           :price-abouve price-abouve-pred
                           :price-below price-below-pred))

        constraint-pairs-A (map (fn [x]
                                  [(find-lookup-fn (first x)) (second x)])
                                constraint-pairs)

        lookupfn-fns-A (map (fn [x]
                               (fn [y]
                                 (lookupfn y ((first x) (second x)))))
                            constraint-pairs-A)]

    ;; apply all fns with args
    (apply concat ((apply juxt lookupfn-fns-A)
                   files))))
```

After the input arguments and function precondition, we read the function from the `let` expression:

- `constraint-pairs`: This breaks function parameters into pairs
- `find-lookup-fn`: This creates a function to look up a corresponding predicate function for a given constraint key (for example, `:time-after` and `:price-above`)
- `constraint-pairs-A`: This replaces a constraint key with the corresponding predicate function
- `lookupfn-fns-A`: This returns another list of functions that call `lookupfn` with the file list, predicate, and submitted value that we've just looked up

The juxt higher order function

The `juxt` function is a Clojure core function that, like `comp`, returns another function that comprises its input functions. The `juxt` function, however, generates a function that is the juxtaposition of its input functions. It returns a vector of the result of applying each function to an input argument. You can supply as many input arguments as you like to a generated function. Just be sure that the source function(s) you supply will operate on the inputs you've provided:

```
(juxt inc dec)
;; #object[clojure.core$juxt$fn__4510 0x1d3be6d "clojure.core$juxt$fn__4510@1d3be6d"]

((juxt inc dec keyword) 10)
;; (11 9 nil)

((juxt inc dec (comp keyword name str)) 10)
;; [11 9 :10]

((juxt inc dec (comp keyword name str)) 10 20 30)
;; clojure.lang.ArityException: Wrong number of args (3) passed to: core/inc

((juxt + *) 10 20 30)
;; [60 6000]

(apply (juxt min max) '(10 20 30))
;; [10 30]
```

With that understanding, let's take a look at the double `apply` expression.

```
(apply concat ((apply juxt lookupfn-fns-A)
               files))))
```

This expression can be explained as follows:

- The `(apply juxt lookupfn-fns-A)` expression will take our list of functions and generate a juxtaposition function
- The generated `juxtaposition` function is then called with the input files (`files`)
- What's returned from this call is the result list of each function in `lookupfn-fns-A` that's inside a list (a list of lists)
- We can normally `concat` many lists together, for example, `(concat [1 2 3] [4 5 6] [7 8 9]) ;; (1 2 3 4 5 6 7 8 9)`
- However, we reach for `apply` to use functions on arguments in lists, for example, `(apply concat [[1 2] [3 4]]) ;; (1 2 3 4)`
- Thus, we `(apply concat ...)` to our list of lists

These are examples of how we can use the lookup function that's developing. Replace the stringified time values, such as (example `2015-08-15T17:18:51.352-00:00`), with ones that reflect what's on your system:

```
(def many-files (file-seq (io/file "data/")))

(lookup :time #inst "2015-08-15T17:18:51.352-00:00")
(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00")
(lookup :time-before #inst "2015-08-15T17:19:00.000-00:00")
(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00"   :time-before #inst "2015-08-15T17:19:00.000-00:00")

(lookup :price 4.028309189176084)
(lookup :price-abouve 12)
(lookup :price-below 12)
(lookup :price-abouve 12 :price-below 20)

(lookup :time-after #inst "2015-08-15T17:18:00.000-00:00"
        :time-before #inst "2015-08-15T17:19:00.000-00:00"
        :price-abouve 12
        :price-below 20)

(lookup :source many-files
        :time #inst "2015-08-15T17:18:51.352-00:00")

(lookup :source "data/"
        :time #inst "2015-08-15T17:18:51.352-00:00")
```

Separate OR AND lookups

Our first pass at a lookup function looks fruitful. We can call it with a number of constraints and get back expected results. However, this function assumes that all our conditions are logical ORs This means a function should satisfy the `a OR b OR c` condition. But this will satisfy only some of our needs. Often, we'll want more than one constraint to be satisfied at the same time. This is a logical AND. Let's implement this function now. While we're at it, we can refactor our first lookup function and reuse some of its parts:

```
(defn generate-input-list [constraints]
  (if (some #{:source} constraints)
    (let [source-source (comp (partial filter #(= :source (first     %1)))
                              (partial partition 2))
          source-value (comp second source-source)
          source-key (comp first source-source)]

      (if (string? source-key)
        (load-directory (source-key constraints))
        source-value))

    (load-directory "data/")))

(defn generate-constraint-pairs [constraints]
  (->> constraints
       (partition 2)
       (remove #(= :source (first %)))))

(defn find-lookup-fn [inp]
  (case inp
    :time specific-time-pred
    :time-after time-after-pred
    :time-before time-before-pred
    :price specific-price-pred
    :price-abouve price-abouve-pred
    :price-below price-below-pred))

;; refactor some code to clean up
(defn lookup-refactored [& constraints]

  ;; ensure constraints are in pairs -> Preconditions
  {:pre [(even? (count constraints))]}
```

```
    ;; map over pairs - find predicate fn based on keyword -   partially apply fn with arg
    (let [files (generate-input-list constraints)
          constraint-pairs (generate-constraint-pairs constraints)

          constraint-pairs-A (map (fn [x]
                      [(find-lookup-fn (first x))                          (second x)])
                                  constraint-pairs)

          lookupfn-fns-A (map (fn [x]
                                 (fn [y]
                      (lookupfn y ((first x) (second                       x)))))
                              constraint-pairs-A)]

      ;; apply all fns with args
      (apply concat ((apply juxt lookupfn-fns-A)
                     files))))

  (defn lookup-and [& constraints]

    ;; ensure constraints are in pairs -> Preconditions
    {:pre [(even? (count constraints))]}

    ;; map over pairs - find predicate fn based on keyword -   partially apply fn with arg
    (let [files (generate-input-list constraints)
          constraint-pairs (generate-constraint-pairs constraints)

          constraint-pairs-B (map (fn [x]
                      [(find-lookup-fn (first x))                          (second x)])
                                  constraint-pairs)

          constraint-predicates (map (fn [x]
                                       ((first x) (second x)))
                                     constraint-pairs-B)

          ;; lookupfn
          ;; constraint-predicates ;; (f1 f2 ...)
          ;; files

          pred-fn (fn [input-tick]
                    (every? (fn [pred]
                              (pred input-tick))
                            constraint-predicates))]

    (lookupfn files pred-fn)))
```

The `generate-input-list`, `generate-constraint-pairs`, and `find-lookup-fn` functions are pulled out of the `let` block from the original `lookup` function. The `lookup-refactored` function is now a lot smaller and we can reuse the first three functions in our AND implementation. The `lookup-and` expression also uses the `let` block to create an input file list and pairs of input constraints. The `constraint-pairs-B` function replaces a constraint key with a corresponding predicate function. The `lookup-and` function differs slightly thereon.

The Var, `constraint-predicates`, is a Var of the result of each predicate function applied with a supplied input. Recall that predicates themselves returned yet another function, for example, `(defn time-after-pred [time] #(.after (:last-trade-time %) time))`. So, the result of this will be in the form of a list of predicate functions.

Given that this lookup function implements a logical AND, we want all of the predicates to hold true for each input tick. Therefore, when `lookupfn` is finally called with a file list (files), we want a function that holds true for all conditions. This is where `pred-fn` comes in.

The `pred-fn` function is a function that takes input ticks as its arguments. For each input tick, we apply the `every?` function, for example, `(every? pos? '(10 20 30)) ;; true`. This expression asks "does every one of the predicate functions hold true for this one tick?" A true or false answer determines what's finally returned by `lookupfn`:

```
(count (lookup-or :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20))
;; 2126

(count (lookup-and :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20))
;; 424

(count (lookup-and :time-after #inst "2015-08-15T17:18:00.000-00:00"
                   :time-before #inst "2015-08-15T17:19:00.000-00:00"
                   :price-abouve 20))
;; 47
```

With the logical AND function implemented, we can try it out and compare the results we get from what was returned with our original OR implementation. The same constraints using an OR condition return `2126` ticks, where the use of an AND condition returns only `424` results. Constraining our first AND further (prices that are only above 20) again shrinks the number of ticks that satisfy all constraints.

So far, we've made progress. However, it would be nice to further simplify our overall interface to use either AND or OR. When comparing only a pseudocode outline of code to an OR AND function where each implementation becomes larger, we'd like to implement some features:

```
(defn lookup-combined' [mode & constraints]

  ;; ensure constraints are in pairs -> Preconditions
  {:pre [(even? (count constraints))]}



  ;; map over pairs - find predicate fn based on keyword - partially apply fn with arg
  (let [files (generate-input-list constraints)
        constraint-pairs (generate-constraint-pairs constraints)

        ;; OR
        constraint-pairs-A (map (fn [x]
                                [(find-lookup-fn (first x)) (second x)])
                                 constraint-pairs)

        lookupfn-fns-A (map (fn [x]
                                (fn [y]
                             (lookupfn y ((first x) (second x)))))
                            constraint-pairs-A)

        ;; AND
        constraint-pairs-B (map (fn [x]
                                [(find-lookup-fn (first x)) (second x)])
                                 constraint-pairs)

        constraint-predicates (map (fn [x]
                                     ((first x) (second x)))
                                   constraint-pairs-B)

        pred-fn (fn [x]
                (every? (fn [pfn]
                           (pfn x))
                        constraint-predicates))]

      ;; OR
      (apply concat ((apply juxt lookupfn-fns-A)
                     files))

      ;; AND
      (lookupfn files pred-fn)))
```

Deriving a query language using macros

To be sure, we could refactor the preceding lookup function again and shrink it to a manageable size. In fact, implementing solutions using first-class functions is the preferred approach in 9 out of 10 cases. However, it's useful to know other approaches for the proverbial 10th case. With the preceding function-based approach understood, there are other ways of implementing syntaxes, which are otherwise known as **Domain-specific Languages** (**DSLs**). The most popular approach in Lisp is using a macro.

Since Clojure syntax is made up of data (homoiconic), we can transform it similar to any other piece of data flowing through our system. Macros are facilities that let us perform this code transformation. In this case, it can help us better design a syntax interface. One thing to be aware of is that the evaluation of inputs (that is, expressions) to your macro is deferred. You have complete control as to if and when they will be evaluated based on the nature of the syntax you provide:

```
(defn apply-juxt-helper [lookupfn-fns]
  (apply concat ((apply juxt lookupfn-fns)
                 files)))

(defn choose-constraint [mode files constraint-pairs]

  (if (= :or mode)

     (->> (quote ~constraint-pairs)

          (map (fn [x#]
                 [(find-lookup-fn (first x#)) (second x#)]))

          (map (fn [x#]
                 (fn [y#]
                   (lookupfn y# ((first x#) (second x#))))))

          (apply-juxt-helper))

     (->> (quote ~constraint-pairs)

          (map (fn [x#]
                 [(find-lookup-fn (first x#)) (second x#)]))

          (map (fn [x#]
                 ((first x#) (second x#))))

          (fn [x#]
            (fn [y#]
              (every? (fn [pfn#]
                        (pfn# y#))
                      x#)))

          (lookupfn files))))

(defmacro lookup-combined [mode & constraints]
  {:pre [(even? (count constraints))]}

  (let [files (generate-input-list constraints)
        constraint-pairs (generate-constraint-pairs constraints)]

    (choose-constraint mode files constraint-pairs)))
```

I've changed the syntax slightly to allow for the logical mode to be passed in as the first argument. But the preceding macro that we've created looks similar to the previous lookup functions we made. I've introduced the `->>` thrush (recall that it passes each output to the last parameter of the subsequent form) to streamline the `let` code. The OR and AND blocks are now slightly cleaned up. As stated earlier, though, the evaluation model is different. The body of the macro, including the call out to `choose-constraint` (`apply-juxt-helper` is just a helper function that's needed to fit with the `->>` thrush pattern), is evaluated because the forms are there. What's interesting, though, is that in the body of `choose-constraint`, we return one of two possible forms based on the mode that's passed in. Either form is the code that ultimately gets returned from the macro expansion, then evaluated, and ultimately returned from the macro.

Indeed, Clojure provides the `macroexpand-1` function that performs this very task. It expands the macro down one level (not expanding all nested macros and expressions), showing the code that is returned, which would ultimately get evaluated by your REPL. So, if we macro expand `lookup-combined` using the `:or` mode, it will look a bit different from a macro expanded `lookup-combined` using the `:and` mode. While doing this is possible, it isn't meant to be read so much as it is a demonstration of how your code is transformed and the difference produced for each option:

```
(macroexpand-1 '(lookup-combined :or :time-after #inst "2015-08-
15T17:18:00.000-00:00" :price-abouve 20))

(clojure.core/->>
 '((:time-after #inst "2015-08-15T17:18:00.000-00:00")
   (:price-abouve 20))
 (clojure.core/map
  (clojure.core/fn
   [x__27781__auto__]
   [(edgaru.eight/find-lookup-fn (clojure.core/first x__27781__auto__))
    (clojure.core/second x__27781__auto__)]))
 (clojure.core/map
  (clojure.core/fn
   [x__27781__auto__]
   (clojure.core/fn
    [y__27782__auto__]
    (edgaru.eight/lookupfn
     y__27782__auto__
     ((clojure.core/first x__27781__auto__)
      (clojure.core/second x__27781__auto__))))))
 (edgaru.eight/apply-juxt-helper))

(macroexpand-1 '(lookup-combined :and :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20))

(clojure.core/->>
 '((:time-after #inst "2015-08-15T17:18:00.000-00:00")
   (:price-abouve 20))
 (clojure.core/map
  (clojure.core/fn
   [x__27783__auto__]
   [(edgaru.eight/find-lookup-fn (clojure.core/first x__27783__auto__))
    (clojure.core/second x__27783__auto__)]))
 (clojure.core/map
  (clojure.core/fn
   [x__27783__auto__]
   ((clojure.core/first x__27783__auto__)
    (clojure.core/second x__27783__auto__))))
 (clojure.core/fn
  [x__27783__auto__]
  (clojure.core/fn
   [y__27784__auto__]
   (clojure.core/every?
    (clojure.core/fn
     [pfn__27785__auto__]
     (pfn__27785__auto__ y__27784__auto__))
    x__27783__auto__)))
 (edgaru.eight/lookupfn edgaru.eight/files))
```

Now we can use the finalized lookup-combined macro to execute in either logical AND or OR mode:

```
(count (lookup-combined :or :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20))
;; 2126

(count (lookup-combined :and :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20))
;; 1915
```

Let's try a slightly more data-oriented query syntax where we pass in a vector of conditions. This is easily deconstructed in the first `let` block of the macro:

```
(defmacro lookup [query-params]

  ;; ensure constraints are in pairs -> Preconditions
  {:pre [(even? (count (rest query-params)))]}

  ;; map over pairs - find predicate fn based on keyword - partially apply fn with arg
  (let [mode (first query-params)
        constraints (rest query-params)
        files (generate-input-list constraints)
        constraint-pairs (generate-constraint-pairs constraints)]

    (choose-constraint mode files constraint-pairs)))
```

But an approach where the query syntax body itself can become data allows for queries to be stored in separate locations. Now we can execute lookups like this:

```
(count (lookup [:or :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20]))
;; 2126

(count (lookup [:and :time-after #inst "2015-08-15T17:18:00.000-00:00" :price-abouve 20]))
;; 1915
```

These are very rudimentary initial steps. We may choose to expand our syntax to include negation or nested AND/OR conditions. Since the evaluation of inputs (that is, expressions) to your macro is deferred, we can pass in constraint expressions instead of values and control if and when these expression paths are chosen.

Summary

These are initial steps to understanding how to read in data using Clojure. We chose to use a filesystem. However, in the real world, data can come from a web source, e-mail source, data feeds from your financial provider, and so on.

Once data was in your system, we took a look at a few ways of querying it. This included simple filtering, then adding more constraints in a logical OR and AND fashion. Most of these approaches involved first-class function manipulations, which is often the preferred choice. However, if you need a syntax to alter the control flow of a program or control if or when inputs get evaluated, custom syntaxes that are provided by macros are a popular approach in Clojure.

In the final chapter, we'll take all the knowledge we've gained and design buy or sell signals based on our analysis so far.

9
Building Algorithms – Strategies to Manipulate and Compose Functions

The ultimate goal of the kind of analytics system we've been building is to get clear signals for the direction of an asset (stocks, in this case). With regard to our SMAs, EMAs and Bollinger Bands that capture market dynamics, let's see if we can use these analyses to glean whether an underlying asset will go up or down.

This chapter will use all the knowledge we've gained so far to design these signals. We'll see how we can reason about this process, and then encode new business rules on top of existing ones to create new functionality and information. In this chapter, we'll cover the following topics:

- Structuring our data for further analysis
- The third refactor of our analytic functions
- Signals using moving averages
- The Relative Strength Index
- Bollinger Band signals

Structuring our data for further analysis

SMAs and EMAs lag an underlying stock price, but in slightly different ways. The SMA assigns equal weight to all input ticks, whereas the EMA assigns greater weight to more recent stock ticks. For example, if we take a baseline time series along with its SMAs and EMAs, we can look out for when the EMA overlaps (above or below) the SMA.

Create the `src/edgar/signals.clj` file and add this namespace definition at the top:

```
(ns edgar.signals
  (:require [edgaru.nine.core :as core]
            [edgaru.nine.analytics :as analytics]
            [edgaru.nine.datasource :as datasource]))
```

To make these comparisons, we'll first have to make a function that joins all our data into one list. Review the `join-averages` function; we'll go through it step by step here:

```
(defn join-averages
  ([tick-window tick-list]

   (let [sma-list (simple-moving-average nil tick-window tick-list)
         ema-list (exponential-moving-average nil tick-window tick-list sma-list)]
     (join-averages tick-list sma-list ema-list)))

  ([tick-list sma-list ema-list]

   (let [trimmed-ticks (drop-while #(not (= (:last-trade-time %)
                                            (:last-trade-time (first sma-list))))
                                   tick-list)]

     (map (fn [titem sitem eitem]

            (if (and (and (not (nil? (:last-trade-time sitem)))
                          (not (nil? (:last-trade-time eitem))))
                     (= (:last-trade-time titem) (:last-trade-time sitem) (:last-trade-time eitem)))

              {:last-trade-time (:last-trade-time titem)
               :last-trade-price (if (string? (:last-trade-price titem))
                                   (read-string (:last-trade-price titem))
                                   (:last-trade-price titem))
               :last-trade-price-average (:last-trade-price-average sitem)
               :last-trade-price-exponential (:last-trade-price-exponential eitem)}

              nil))

          trimmed-ticks
          sma-list
          ema-list))))
```

The `join-averages` function overlays a stock price tick list, SMA list, and an EMA list into one list. Recall that the moving average values begin 20 ticks (or whatever you set the tick window to be) into the source tick list. As such, the `join-averages` function will need to line up all three lists to the same trade time. This is what the `trimmed-ticks` Var represents. We've created it using the `drop-while` function. This function takes a collection and from left to right keeps dropping its elements until some condition is met. This condition is supplied by the predicate we've provided. Since we know that `tick-list` will be the longer list, we'll drop its earliest elements (elements on the left) until it's `:last-trade-time` matches `:last-trade-time` from the first `simple-moving-average`.

We can now map over all three lists at once. Remember that the `map` function accepts as many list as you want to give it. The number of parameters to the mapping function must match the number of lists that are supplied. The `map` function will also process all lists until the first list is exhausted. In our mapping function, the `if` block on each iteration will ensure that `:last-trade-time` in all lists are equal:

```
(and (and (not (nil? (:last-trade-time sitem)))
                (not (nil? (:last-trade-time eitem))))
          (= (:last-trade-time titem) (:last-trade-time sitem) (:last-trade-time eitem)))
```

We then simply return a `map` with all the relevant data elements inside it:

```
{:last-trade-time (:last-trade-time titem)
 :last-trade-price (if (string? (:last-trade-price titem))
                     (read-string (:last-trade-price titem))
                     (:last-trade-price titem))
 :last-trade-price-average (:last-trade-price-average sitem)
 :last-trade-price-exponential (:last-trade-price-exponential eitem)}
```

I've introduced a subtle code change that we must address. `simple-moving-average` and `exponential-moving-average` (and `bollinger-band`) compute their final values with the `reduce` function. The `reduce` function has a finality to it, in that it returns a concrete list, which is no longer lazy. Inversely, if you execute `reduce` over a lazy sequence, it will run indefinitely. The `join-averages` function and all subsequent signal functions will be lazy and expect infinite sequences. Back in *Chapter 3, Developing the Simple Moving Average*, we learned that while data is being processed, a good architectural principle is to keep it in lazy sequences. As long as everybody's using lazy sequences, we can treat these as data flow pipelines that easily compose functions and pieces of functions. So, now is a good time to return to our core analytic functions and make them lazy.

A third refactor of the analytic functions

We'll refactor `simple-moving-average`, `exponential-moving-average`, and `bollinger-band` at the same time. At each point, we need to take the `reduce` block of code and change it to a lazy version that fits a function's process criteria:

```
(defn simple-moving-average
  [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]

    (map (fn [ech]

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

                 taverage (/ tsum (count ech))]

             (merge
              (zipmap etal-keys
                      (map #(% (last ech)) etal-keys))
              {output-key taverage
               :population ech})))
         (partition tick-window 1 tick-list))))
```

The `simple-moving-average` function partitions an input tick list into a sliding window of tick data, incrementing by one tick. We now have a list of lists as input to the original `reduce` function. It then builds a new list using `lazy-cat` for each new computation to the end of a result list. However, doing this is unnecessary as each element in the list doesn't need to know about the other. We can simply `map` over the partitioned list and return each calculation independently:

```
(defn exponential-moving-average
  ([options tick-window tick-list]
   (exponential-moving-average options tick-window tick-list (simple-moving-average {} tick-window tick-list)))
  ([options tick-window tick-list sma-list]

   ;; 1. calculate 'k'
   ;; k = 2 / N + 1
   ;; N = number of days
   (let [k (/ 2 (+ tick-window 1))

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

     ;; 2. get the simple-moving-average for a given tick - 1
     (last (reductions (fn [rslt ech]

                         ;; 3. calculate the EMA ( for the first tick, EMA(yesterday) = MA(yesterday) )
                         (let [;; price(today)
                               ltprice (input-key ech)

                               ;; EMA(yesterday)
                               ema-last (if (output-key (last rslt))
                                          (output-key (last rslt))
                                          (input-key ech))

                               ;; ** EMA now = price(today) * k + EMA(yesterday) * (1 - k)
                               ema-now (+ (* k ltprice)
                                          (* ema-last (- 1 k)))]

                           (lazy-cat rslt
                                     [(merge
                                       (zipmap etal-keys
                               (map #(% (last (:population ech))) etal-keys))
                                       {output-key ema-now})])))
                       '()
                       sma-list)))))
```

The process criteria for `exponential-moving-average` is slightly different as the calculation of the current iteration needs the last value from the previous calculation. So, `map` won't work in this case. The `reduce` function provides access to an accumulation of results, but it isn't lazy. This is where we get to use `reductions` again. Recall that `reductions` behaves like `reduce`, but in a lazy fashion. The `reduce` function folds over a list and maintains a running result while iterating over each element, whereas `reductions` returns a lazy sequence of intermediate results along with the final result. Therefore, the only change made to the algorithm is swapping out `reduce` for `reductions` and taking out the `last` item (the accumulated list) from the result:

```
(defn bollinger-band
  ([tick-window tick-list]
  (bollinger-band tick-window tick-list (simple-moving-average nil tick-window tick-list)))

  ([tick-window tick-list sma-list]

  ;; At each step, the Standard Deviation will be: the square root of the variance (average of the squared differences from
the Mean)
   (map (fn [ech]

          (let [;; get the Moving Average
                ma (:last-trade-price-average ech)

                ;; work out the mean
                mean (/ (reduce (fn [rlt ech]
                                  (+ (:last-trade-price ech)
                                     rlt))
                                0
                                (:population ech))
                        (count (:population ech)))

          ;; Then for each number: subtract the mean and square the result (the squared difference)
                sq-diff-list (map (fn [ech]
                             (let [diff (- mean (:last-trade-price ech))]
                                   (* diff diff)))
                                  (:population ech))

              variance (/ (reduce + sq-diff-list) (count (:population ech)))
                standard-deviation (. Math sqrt variance)]
            {:last-trade-price (:last-trade-price ech)
             :last-trade-time (:last-trade-time ech)
             :upper-band (+ ma (* 2 standard-deviation))
             :lower-band (- ma (* 2 standard-deviation))}))
        sma-list)))
```

The `bollinger-band` refactor follows the same pattern as the one used for the `simple-moving-average` refactor. Though each item is iterated over, it doesn't need to know about any other item. Thus, we can simply `map` over the source `simple-moving-average` list, returning a map of the `:last-trade-price`, `:last-trade-time`, `:upper-band`, and `:lower-band` entries.

Signals using moving averages

Now that our analytic functions are lazy and we can join them into one list, we can start thinking about what signals we may be interested in. A simple and direct signal is finding out whether the EMA in one tick element has crossed above the SMA from a previous element. Another signal would be the inverse of this—we can find out whether, in the first element, the EMA has crossed below the SMA from the second element. This is another process criteria where a current tick needs to know about a previous tick. However, in this situation, the previous computation isn't needed by the current one. We can just partition our ticks by two to check for a crossover. Let's take a look at an implementation now:

```
(defn moving-averages-signals
  "Takes baseline time series, along with 2 other moving averages.
  Produces a list of signals where the 2nd moving average overlaps (abouve or below) the first.
  By default, this function will produce a Simple Moving Average and an Exponential Moving Average."

  ([tick-window tick-list]

   (let [sma-list (simple-moving-average nil tick-window tick-list)
         ema-list (exponential-moving-average nil tick-window tick-list sma-list)]
     (moving-averages-signals tick-list sma-list ema-list)))

  ([tick-list sma-list ema-list]

  ;; create a list where i) tick-list ii) sma-list and iii) ema-list are overlaid
   (let [joined-list (join-averages tick-list sma-list ema-list)
         partitioned-join (partition 2 1 (remove nil? joined-list))]

    ;; find time points where ema-list (or second list) crosses over the sma-list (or 1st list)
     (map (fn [[fst snd]]
            (let [
              ;; in the first element, has the ema crossed abouve the sma from the second element
              signal-up (and (< (:last-trade-price-exponential snd) (:last-trade-price-average snd))
                             (> (:last-trade-price-exponential fst) (:last-trade-price-average fst)))

              ;; in the first element, has the ema crossed below the sma from the second element
            signal-down (and (> (:last-trade-price-exponential snd) (:last-trade-price-average snd))
                             (< (:last-trade-price-exponential fst) (:last-trade-price-average fst)))

                  raw-data fst]

           ;; return either i) :up signal, ii) :down signal or iii) nothing, with just the raw data
              (if signal-up
                (assoc raw-data :signals [{:signal :up
                                          :why :moving-average-crossover
                                               :arguments [fst snd]}])
                (if signal-down
                  (assoc raw-data :signals [{:signal :down
                                            :why :moving-average-crossover
                                                 :arguments [fst snd]}])
                  raw-data))))
          partitioned-join))))
```

In the preceding code, `moving-averages-signals` will generate a simple and exponential moving average if it hasn't been passed in. In the second `let` block, we'll join the underlying ticks, SMA, and EMA into one list (called `joined-list`). As described previously, we then partition this `joined-list` into twos (called `partitioned-join`).

Now we `map` over `partitioned-join` and apply our algorithm. The mapping function takes each element and destructures it into the first and second elements (`fst` and `snd`). Recall from *Chapter 4, Strategies for Calculating and Manipulating Data,* that destructuring is a small custom language used to extract values from data structures. In this case, each element in `partitioned-join` will be a collection and not an associative structure (a map). So, `[[fst snd]]` is the list destructuring syntax needed to extract elements 0 and 1 and bind them to `fst` and `snd`. We can now use these Vars to see whether the second exponential price is below the second average price and the first exponential price is above the first average price. We then assign the true or false result to `signal-up`:

```
(and (< (:last-trade-price-exponential snd) (:last-trade-price-average snd))
     (> (:last-trade-price-exponential fst) (:last-trade-price-average fst)))
```

We will now make the inverse check and assign this true or false result to `signal-down`:

```
(and (> (:last-trade-price-exponential snd) (:last-trade-price-average snd))
     (< (:last-trade-price-exponential fst) (:last-trade-price-average fst)))
```

As these calculations are contingent on the intersections between the ticks, either the first or second tick can be considered a signal point. I've chosen the first (assigned to `raw-data`) as any algorithm would see this signal sooner than later. However, this choice will have to be remembered for any future signals that we'd like to line up.

The last expressions we'll look at are nested `if` statements. Logically, we know that `signal-up` and `signal-down` are mutually exclusive. So, if `signal-up` is true, we can return raw data with an `:up` signal attached to it along with any other information such as the reason. If `signal-up` is false and `signal-down` is true, then we'll return a `:down` signal and any other relevant data. If neither is true, we can just return `raw-data`:

```
(if signal-up
  (assoc raw-data :signals [{:signal :up
                             :why :moving-average-crossover
                             :arguments [fst snd]}])
  (if signal-down
    (assoc raw-data :signals [{:signal :down
                               :why :moving-average-crossover
                               :arguments [fst snd]}])
    raw-data)
```

The Relative Strength Index

Our next signal is based on the Bollinger Band, but this signal uses yet another signal called the **Relative Strength Index** (**RSI**) divergence. RSI is a technical momentum indicator that indicates the strength or weakness of a stock based on recent trading. It compares the size of recent gains to recent losses in order to determine whether an asset is overbought or oversold. An RSI divergence signal can appear before the momentum or trend of stock changes. This provides a valuable early warning mechanism and/or a confirming signal to associated trend changes. In this analysis, we'll take a look at a few things:

- Does the price make a higher high from previous price points?
- Does RSI divergence make a lower high?
- Does divergence take place above the overbought line?
- For an entry signal, check whether one of the next three closes are underneath the priors (or the same holds true in the opposite direction)

Review this code; we'll go through it step by step:

```
(defn relative-strength-index

  [tick-window tick-list]

  (let [twindow (or tick-window 14)
        window-list (partition twindow 1 tick-list)]
```

```
    ;; run over the collection of populations
    (last (reductions (fn [rslt ech]

                   ;; each item will be a population of tick-window (default of 14)
                         (let [pass-one (reduce (fn [rslt ech]

                                                  (let [fst (:last-trade-price (first ech))
                                                snd (:last-trade-price (second ech))

                                                             up? (< fst snd)
                                                      down? (> fst snd)
                                                sideways? (and (not up?) (not down?))]

                                                          (if (or up? down?)
                                                            (if up?
                                                (conj rslt (assoc (first ech) :signal :up))
                                                (conj rslt (assoc (first ech) :signal :down)))
                                              (conj rslt (assoc (first ech) :signal :sideways)))))
                                                      []
                                                (partition 2 1 (remove nil? ech)))


                            up-list (:up (group-by :signal pass-one))
                            down-list (:down (group-by :signal pass-one))

                               avg-gains (/ (apply +
                                          (map :last-trade-price up-list))
                                              tick-window)
                               avg-losses (/ (apply +
                                           (map :last-trade-price down-list))
                                               tick-window)

                               rs (if-not (= 0 avg-losses)
                                     (/ avg-gains avg-losses)
                                     0)
                               rsi (- 100 (/ 100 (+ 1 rs)))]

                   (conj rslt {:last-trade-time (:last-trade-time (first ech))
                               :last-trade-price (:last-trade-price (first ech))
                                        :rs rs
                                        :rsi rsi})))
                     []
                     window-list))))
```

The `relative-strength-index` function implements raw RSI, a momentum oscillator that measures the speed and change of price movements. It oscillates between 0 and 100. We describe the function, beginning with function arguments, if no `tick-window` is given, it defaults to 14, which is the standard RSI measurement period.

1. The first step is to `partition` by 14 our source `tick-list`, incrementing each new list by one (`window-list`).
2. We then perform a lazy reduce, called `reductions`, over the partitioned list of lists.
3. Next up is the `let` block in the reducing function, where we first perform a second `reduce` on a two-window partition. We partition by two as, for each pair, we will check if the second price is greater, less than, or equal to the first (`up?`, `down?`, or `sideways?`). This is just the first pass of the algorithm (`pass-one`).
4. The structure of `pass-one` will be similar to `tick-list`, with additional `:signal` entries of `:up`, `:down`, or `:sideways`. We can now group `pass-one` by (`group-by`), which is the `:signal` key, pulling out the `:up` and `:down` signals (`up-list` and `down-list`).
5. Summing up `:last-trade-price` from all the :up movements `(apply + (map :last-trade-price up-list))`, and then dividing by a `tick-window` of 14 will give us the average gains (`avg-gains`).
6. We will perform the same calculation, though for the average losses this time (`avg-losses`).
7. We can now calculate the relative strength of the stock over a given 14-tick period by dividing the gains by the losses `(/ avg-gains avg-losses)` (provided there are no zero losses).
8. From this relative strength, we just need to factor it by 100 `(- 100 (/ 100 (+ 1 rs)))` to get the RSI (`rsi`).

So, if we had average gains of 20 and average losses of 15, the RSI would be `(- 100 (/ 100 (+ 1 (/ 20 15)))) ;; 400/7` or 57.14285714285714. Otherwise, if we had average gains of 15 and average losses of 20, the RSI would be `(- 100 (/ 100 (+ 1 (/ 15 20)))) ;; 300/7` or 42.85714285714286. Therefore, for a given tick list, we will have a running calculation of RSIs, and also have an indication as to when the average gains will change relative to the average losses.

Bollinger Band signals

Bollinger Band signals involve a few more checks. When the band width is very low, this indicates that the price will break out sooner than later. For this strategy, let's call it "A", we need to check for a few things:

- Is the moving average in an up or down market?
- Width-wise, we should check for a narrowing Bollinger Band. However, is the bandwidth less than the three previous narrow bands? Three is a random number that can be tuned once this algorithm has been tested on real data.
- Is the close outside the Bollinger Bands, and previous high or low swing inside the bands?

When the width of the Bollinger Band is high (indicating high volatility), this can mean that a trend is ending soon or that it's either going to change direction or consolidate. For this strategy, let's call it "B", we need to check a different set of conditions:

- Is the moving average in a sideways (or choppy) market? Here, we will check whether the last three closes are above or below the Bollinger Bands.
- Width-wise, we should check for a widening Bollinger Band. Is the band width greater than the three previous bands? Three is a random number that can be tuned once this algorithm has been tested on real data.
- Here, we will apply the RSI divergence, that is, the change of the average gains, relative to the average losses.

Take a look at the following `bollinger-band-signals` function. We will go through it in detail afterwards:

```
(declare sort-bollinger-band)
(defn bollinger-band-signals

  ([tick-window tick-list]
   (let [sma-list (simple-moving-average nil tick-window tick-list)
         bband (bollinger-band tick-window tick-list sma-list)]

     (bollinger-band-signals tick-window tick-list sma-list bband)))

  ([tick-window tick-list sma-list bband]

   (last (reductions (fn [rslt ech-list]

                       (let [
                   ;; track widest & narrowest band over the last 'n' ( 3 ) ticks
                           sorted-bands (sort-bollinger-band ech-list)
                             most-narrow (take 3 sorted-bands)
                             most-wide (take-last 3 sorted-bands)

                         partitioned-list (partition 2 1 (remove nil? ech-list))

                         upM? (up-market? 10 (remove nil? partitioned-list))
                         downM? (down-market? 10 (remove nil? partitioned-list))

                             side-market? (and (not upM?)
                                               (not downM?))

                             ;; find last 3 peaks and valleys
                         peaks-valleys (find-peaks-valleys nil (remove nil? ech-list))
                           peaks (:peak (group-by :signal peaks-valleys))
                         valleys (:valley (group-by :signal peaks-valleys))]


                         (if (empty? (remove nil? ech-list))

                           (conj rslt nil)

                           (if (or upM? downM?)

                             ;; A.
                   (calculate-strategy-a rslt ech-list most-narrow upM? peaks valleys)

                             ;; B.
                   (calculate-strategy-b rslt ech-list most-wide peaks valleys)))

                         (conj rslt (first ech-list))))
                     []
                     (partition tick-window 1 bband)))))
```

The `bollinger-band-signals` and accompanying helper and calculation functions are rather large, so we'll tackle these one section at a time. The signal function needs a tick list and an accompanying SMA and Bollinger Band. These are passed in or are otherwise generated:

- Before each strategy is implemented, we will first collect the three widest and narrowest bands over the last 20 ticks (`most-narrow` and `most-wide`). The `sort-bollinger-band` function is defined as follows. At each tick, we will only take the difference between the upper and lower bands and then sort the result using this difference:

```
(defn sort-bollinger-band [bband]
  (let [diffs (map (fn [inp]
               (assoc inp :difference (- (:upper-band inp) (:lower-band inp))))
                     (remove nil? bband))]
    (sort-by :difference diffs)))
```

- Similar to the `relative-strength-index` function, we `partition` the source `tick-list` by two in order to determine whether there's an up market or down market (`upM?` or `downM?`). This calculation is slightly different, however, in that we determine whether every new tick is greater than the previous one over a 10-tick period (`upM?`):

```
(defn up-market? [period partitioned-list]
  (every? (fn [inp]
            (< (:last-trade-price (first inp))
               (:last-trade-price (second inp))))
          (take period partitioned-list)))
```

- We'll perform the same, but inverse calculation to determine the down market (`downM?`), over a 10-tick period:

```
(defn down-market? [period partitioned-list]
  (every? (fn [inp]
            (> (:last-trade-price (first inp))
               (:last-trade-price (second inp))))
          (take period partitioned-list)))
```

- The next set of calculations finds the peaks and valleys in the source tick list. A peak is a set of three prices where we begin at any price level, go upward, then downward. The opposite of this is applied to a valley (`peaks-valleys`). Next, we'll group and collect each of both (`peaks` and `valleys`):

```
(defn find-peaks-valleys [options tick-list]


  (let [{input-key :input
         :or {input-key :last-trade-price}} options]

    (reduce (fn [rslt ech]
              (let [fst (input-key (first ech))
                    snd (input-key (second ech))
                    thd (input-key (nth ech 2))

             valley? (and (and (-> fst nil? not) (-> snd nil? not) (-> thd nil? not))
                          (> fst snd)
                          (< snd thd))
             peak? (and (and (-> fst nil? not) (-> snd nil? not) (-> thd nil? not))
                        (< fst snd)
                        (> snd thd))]

                (cond
                peak? (conj rslt (assoc (second ech) :signal :peak))
                Valley? (conj rslt (assoc (second ech) :signal :valley)))
                :else rslt)))
            []
            (partition 3 1 tick-list))))
```

With the first `let` block out of the way, we will now either apply strategy A or B based on whether we have an up, down, ("A") or sideways market ("B").

To calculate strategy A, we will check for a narrowing band. Is the band width less than the three previous narrow bands? Is the close outside the Bollinger Band? Were there previous high or low swings inside the band?

```
(defn calculate-strategy-a [rslt ech-list most-narrow upM? peaks valleys]

(let [latest-diff (- (:upper-band (last ech-list)) (:lower-band (last ech-list)))
       less-than-any-narrow? (some (fn [inp] (< latest-diff (:difference inp))) most-narrow)]

    (if less-than-any-narrow?

      (if upM?

       (if (and (< (:last-trade-price (last ech-list)) (:upper-band (last ech-list)))
                (> (:last-trade-price (last peaks))
                 (:upper-band (last (some #(= (:last-trade-time %)
                                  (:last-trade-time (last peaks)))
                                          ech-list)))))

           (conj rslt (assoc (last ech-list) :signals [{:signal :down
                                                        :why :bollinger-close-abouve
                                                  :arguments [ech-list peaks]}]))

           (conj rslt (last ech-list)))
```

```
          (if (and (> (:last-trade-price (last ech-list)) (:lower-band (last ech-list)))
                   (< (:last-trade-price (last valleys))
                      (:lower-band (last (some #(= (:last-trade-time %)
                                                   (:last-trade-time (last valleys)))
                                               ech-list)))))

              (conj rslt (assoc (last ech-list) :signals [{:signal :up
                                                           :why :bollinger-close-below
                                                           :arguments [ech-list valleys]}]))

              (conj rslt (last ech-list)))))))
```

- Of our overall 20 tick window, we have the three most narrow bands (`most-narrow`). Using these, we will determine whether the latest difference in Bollinger Bands is less than the last three most narrow bands (`less-than-any-narrow?`).
- If the latest difference is less than the previous three most narrow bands, *and* we're in an up market, then we can determine the exit or `:down` signal. We will also determine the close outside (above) the upper band, and whether the previous swing is high inside the band:

```
(and (< (:last-trade-price (last ech-list)) (:upper-band (last ech-list)))
     (> (:last-trade-price (last peaks))
        (:upper-band (first (some #(= (:last-trade-time %)
                                      (:last-trade-time (last peaks)))
                                      ech-list)))))
```

If the last condition is true, we will return a down signal. Otherwise, we will just return the latest tick:

```
{:signal :down
 :why :bollinger-close-abouve
 :arguments [ech-list peaks]}
```

If the latest difference is less than the last three most narrow bands, *and* we're in a down market, then we make a determination on the entry or `:up` signal. Is the close outside (below) the lower band, and is the previous swing low inside the band?

```
(and (> (:last-trade-price (last ech-list)) (:lower-band (last ech-list)))
     (< (:last-trade-price (last valleys))
        (:lower-band (last (some #(= (:last-trade-time %)
                                     (:last-trade-time (last valleys)))
                                     ech-list)))))
```

If this is the case, we return an up signal. Otherwise, we just return the latest tick:

```
{:signal :up
 :why :bollinger-close-below
 :arguments [ech-list valleys]}
```

This concludes strategy A. Strategy "B" is slightly more involved, where we employ the RSI. Determining the set of conditions that are true is the minutiae that we will look at here:

```
(defn calculate-strategy-b [rslt ech-list most-wide peaks valleys]

  (let [latest-diff (- (:upper-band (last ech-list)) (:lower-band (last ech-list)))
        more-than-any-wide? (some (fn [inp] (> latest-diff (:difference inp))) most-wide)]

    (if more-than-any-wide?

      ;; B iii RSI Divergence
      (let [
            OVER_BOUGHT 80
            OVER_SOLD 20
            rsi-list (relative-strength-index 14 ech-list)


            ;; i. price makes a higher high and
            higher-highPRICE? (if (empty? peaks)
                                false
                                (> (:last-trade-price (last ech-list))
                                   (:last-trade-price (last peaks))))


            ;; ii. rsi divergence makes a lower high
            lower-highRSI? (if (or (empty? peaks)
                                (some #(nil? (:last-trade-time %)) rsi-list)
                                   #_(not (nil? rsi-list))
                                )
                             false
                             (< (:rsi (last rsi-list))
                                (:rsi (last (filter (fn [inp]
                                                      (= (:last-trade-time inp)
                                                 (:last-trade-time (last peaks))))
                                                  rsi-list)))))

           ;; iii. and divergence should happen abouve the overbought line
            divergence-overbought? (> (:rsi (last rsi-list))
                                     OVER_BOUGHT)



            ;; i. price makes a lower low
            lower-lowPRICE? (if (or (empty? valleys)
                            (some #(nil? (:last-trade-time %)) rsi-list))
                                false
                                (< (:last-trade-price (last ech-list))
                               (:last-trade-price (last valleys))))

            higher-highRSI? (if (or (empty? valleys)
                                    (not (nil? rsi-list)))
                              false
                              (> (:rsi (last rsi-list))
                                 (:rsi (last (filter (fn [inp]
                                             (= (:last-trade-time inp)
                                         (:last-trade-time (last valleys))))
                                                       rsi-list)))))

            divergence-oversold? (< (:rsi (last rsi-list))
                                    OVER_SOLD)]

       (if (and higher-highPRICE? lower-highRSI? divergence-overbought?)

          (conj rslt (assoc (last ech-list) :signals [{:signal :down
                                   :why :bollinger-divergence-overbought
                                   :arguments [peaks ech-list rsi-list]}]))

       (if (and lower-lowPRICE? higher-highRSI? higher-highRSI? divergence-oversold?)

            (conj rslt (assoc (last ech-list) :signals [{:signal :up
                                     :why :bollinger-divergence-oversold
                                                            :arguments
[valleys ech-list rsi-list]}]))

            (conj rslt (last ech-list)))))

      (conj rslt (last ech-list)))))
```

Starting with the first `let` block, we can determine whether the latest Bollinger Band difference is wider than the three widest points in the last 20 ticks (`more-than-any-wide?`). If it is, then we can apply our algorithm:

- We first set the `OVER_BOUGHT` and `OVER_SOLD` levels

- Then, for a given 20-tick list (`ech-list`), calculate a list of 14-tick windows that are relative to the strength indices (`rsi-list`)

If the latest price is higher than the last peak, then we have a price that is higher than the last high price (`higher-highPRICE?`). We need two other, accompanying metrics:

- Is the latest RSI less than the latest (`time-corresponding`) tick from our peaks? If so, then we have a lower high RSI (`lower-highRSI?`).
- Is the latest RSI above the `OVER_BOUGHT` line? If so, then we have an RSI divergence that is overbought (`divergence-overbought?`).

If the latest price is lower than the last valley, then we have a price that is lower than the last low price (`lower-lowPRICE?`). Again, we'll need two other accompanying metrics:

- Is the latest RSI greater than the latest (`time-corresponding`) tick from our valleys? If so, then we have a higher RSI than our last low tick (`higher-highRSI?`).
- Now if the latest RSI is less than the `OVER_SOLD` level, then we have an RSI divergence that is over sold (`divergence-sold?`).

In effect, over a given range, if we have a convergence of a higher high price, lower high RSI, and the RSI divergence is overbought, then this indicates a `:down` (or sell) signal.

If we have a convergence of a lower low price, higher high RSI, and a divergence that is oversold, then we have an `:up` (or buy) signal.

Summary

The real power of this approach is when we start combining signals. So far, we've only looked at moving averages (lagging indicators) and Bollinger Band signals (leading and confirming indicators). But, there are many more strategies that you can employ to make financial decisions.

What should be emphasized is the focus on taking normal business rules and applying them directly with a set of filters, list transformations, and so on, to yield a system that provides us with valuable business information. We've taken our lagging indicator functions (SMA, EMA, and Bollinger Band) and composed them together to get new information—buy or sell signals.

Index