HIGH STAKES

AUTOMATIC EXCHANGE BETTING

AUTOMATING THE BETTING PROCESS ~ FROM STRATEGY TO EXECUTION

COLIN MAGEE

The past decade has seen a technological revolution in facilities for betting on horseracing and other sports. From the proliferation of online information to the advent of betting exchanges, such as Betfair, which matches 15 times as many daily transactions as the London Stock Exchange, it is now possible to analyse form and betting odds; run selection systems; place bets and track results electronically. Crucially, the availability of programmatic access to exchange markets via an application programming interface (API) enables the final piece in reliable automation of the entire betting process.

Automatic Exchange Betting examines each stage of the betting process and shows how it can be automated, with numerous examples applied to horseracing, all implemented using the Betfair API. Betting strategies, best practice and pitfalls are presented which illustrate the concepts of betting automation, together with practical, re-useable code examples written in open source software to work with Betfair's Free Access API.

A combination of speed of execution, consistency and scalability means hitherto impossible betting strategies become feasible, and methodical approaches to betting can be implemented without manual intervention.

Automatic Exchange Betting introduces an entirely new paradigm for betting.

Colin Magee has worked for international finance, e-commerce and statistical software companies for over 15 years. He is a lifelong horseracing enthusiast (as racegoer, occasional owner and punter), and proprietor of www.betwise.co.uk

AUTOMATIC EXCHANGE BETTING

AUTOMATING THE BETTING PROCESS ~ FROM STRATEGY TO EXECUTION

COLIN MAGEE

HIGH STAKES

First published 2008

High Stakes Publishing
21 Great Ormond Street
London WC1N 3JB

www.highstakespublishing.com

A CIP catalogue record for this book is available from the British Library.

ISBN 978-1-84344-041-3

2 4 6 8 10 9 7 5 3 1

Printed and bound in the UK

Acknowledgements

Thanks to The Betfair API Product Development Team, in particular Tom Johnson, Mark Leavitt and Kirsty Frampton, for their review and support for the book.

To Laura Stevens and *The Racing Post* internet team for permission to use and publish data from *www.racingpost.co.uk* in order to show the generation of an automatic oddsline.

To Sandy Paterson of *www.equiformratings.co.uk* for his comments, and for being a good sport in supplying his ratings for automation and scrutiny in Part 3, *Automated Strategies in Practice*.

Thanks also to those who I contacted in order to pass comment on the example Perl code, including Randal Schwartz for finding time to make some general pointers, and to Mike Houghton, who also developed the Betwise website. Any remaining errors or idiosyncrasies are the author's. Further improvements and additions to code are also welcome at *www.betwise.co.uk*.

Last but not least, I thank my wonderful wife for her support and encouragement during the writing of this book.

CONTENTS

Preface

Structure of This Book

The concepts, code and explanations in the book are presented as far as possible to be accessible to the developer and the non-developer, as well as the experienced and inexperienced bettor. Whilst an existing appreciation of both is definitely helpful, especially if wanting to implement automated betting strategies, it is assumed that an understanding of the whole process is within the grasp of the layperson, with resources and other references provided to delve deeper where required.

The aim is to build up a complete picture of automating betting strategies from first concepts to implementing them by risking real capital. Pitfalls, best practice, ideas for improvement and examples are discussed along the way, corresponding to each part of the automatic betting process.

The book is organised in 3 parts, in order to build up that picture, as described below:

Part 1: Exploring Requirements for Betting Automation

Part 1 provides an overview of the betting process and a discussion of the elements involved. Based on this discussion we offer a logical framework for automating the betting process consisting of key stages. We consider what is required for automation of each stage and the whole strategy, in terms of methods used, an IT environment and data access.

Part 2: Implementing A Framework for Betting Automation

In Part 2 of the book we delve deeper into the specifics of implementing a framework for automatic betting.

Starting with general coverage of the Betfair API (or *Application Programming Interface*, our means of connecting automatically to the Betfair exchange), we go on to describe and implement a specific strategy for automatically generating and then betting an oddsline. This is used as a persistent example for each stage of the betting process. The example code is explained with sufficient depth so as to be suitable for those starting with the Betfair API for the first time, or starting with Perl and Linux, although it is also possible to run most of the code “as is”, or simply to follow the discussion.

We also cover many other possibilities for adapting the code and the framework to suit different betting strategies.

Part 3: Automated Strategies in Practice

In Part 3 we start testing and running our example strategy on a live basis, risking real capital and iteratively improving the implementation.

Subsequently, we introduce a third party set of ratings and tissue prices, then run two strategies in parallel over a two week period, leading to a review of all bets and an examination of successes and failures.

Finally, we look at lessons learned from the implementation and running further automated strategies going forward.

Example Programs and other resources

All the example programs, together with sample output from automated strategies referred to in the text, can be downloaded by registering for free at *www.betwise.co.uk*.

Whilst the programs have been used as the basis for implementing effective automated strategies, they are provided here as examples, not as supported software.

Readers are free to use and adapt the software as they wish, provided that the copyright information remains (full licence terms are included with the examples online). All the code in *Automatic Exchange Betting* is provided on an "as is" basis, and comes with absolutely no warranty whatsoever.

Other useful resources for automatic betting, including updates to code, errata and services such as the *Smartform Racing Database* (the automated form database referenced in Parts 1 and 3), can be found at *www.betwise.co.uk*.

See also the Bibliography for learning resources and the Appendices for notes on installation of example software.

Conventions Used

Italics
Italics are used for urls, the first instance of some names, and proper nouns such as horse names.

`Monospace`
A monospace font is used for code listings and shell commands.

`Monospace italics`
Monospace italic is used for program names, file names, path names and other input to be supplied by the user (e.g. program variables such as *`username`*).

`Bold monospace`
Bold monospace is used for highlighting or emphasising key elements of code, program names and shell commands.

Naming conventions for Betfair API Services and XML elements
Betfair API Services such as **GetMarketPricesCompressed** appear in bold with all words capitalised and no spaces (as per the general Betfair convention).

XML elements from Betfair API Services such as **marketId** also appear in bold as per the general Betfair convention (the first word for an XML element is in lower case with subsequent words capitalised).

Naming conventions for example Perl programs, functions and variables:
Example Perl programs are assigned the *.pl* suffix. Example Perl functions to call Betfair services are listed in the library of functions `BetfairAPI6Examples.pm` in Appendix 2. Each function shares the name of the Betfair API service it is accessing, but the Perl name is in lower case and separated by underscores for each word.

For example, the Betfair API Service **GetMarketPricesCompressed** has a corresponding Perl function in `get_market_prices_compressed`.

Perl variables are usually given the same name as the Betfair XML elements whose content they refer to. For example, **marketId** refers to the Betfair XML element whose content is a Betfair market ID number. The Perl scalar variable `$marketId` or hash value `{'marketId'}` refers to the same content, assigned within the context of a program.

Chapter 4 covers available Betfair API Services and example Perl functions in more depth.

Introduction

It seems that the use of computers, the art of programming and The Sport of Kings were made for each other, right from the beginning.

Charles Babbage's plans for a general purpose, programmable computer in the mid-nineteenth century, known as *The Analytical Engine*, are widely seen as the forerunner of the modern computer.

Ada Lovelace, the daughter of Lord Byron, is accredited with describing the first computer program by virtue of her collaboration with Babbage, specifically in her notes on the "*Sketch of the Analytical Engine"*.

Perhaps less well known is that Babbage tried mathematically handicapping horse races, and using the Analytical Engine for the purpose of profitable betting.

By all accounts, Ada was more passionate if not more obsessed about horseracing than Babbage. When the British government's funding for the Analytical Engine ran out, the pair turned with vigour to finding alternative funding at the racecourse – and failed, miserably. We note, soberly, that attempting to predict outcomes of unknown events is a risky business.

So, using computers for horseracing and betting analysis is an idea that is not new, despite some notable advances in technology and the quality of data available since the mid-nineteenth century, not to mention some successes using analytical methods alone to make profitable selections.

However, the idea of using computers to automate the whole betting process has only been possible in the few years since betting execution has gone online, at the turn of the twenty-first century.

For those who wish to understand or take advantage of this opportunity, there is very little material on how to make this happen or what stages are involved. For those who are not sure what the advantages are, the same is true.

This book seeks to redress that balance by discussing the betting process and showing how to automate it from the ground up, with particular reference to horseracing in the UK and Ireland. The idea of what a "betting process" is can vary, so we provide a framework for the betting process, and cover automation within the context of that framework. (Suffice to say for now, it covers everything from selection of a bet to placement of a bet).

Specifically, readers may find the stages of the framework useful in defining what needs to be done to create and automate their own systems for profit, or, if wishing to concentrate on automation in one area only (for example, if already happy with a selection method but wanting to automate staking and execution), may find a useful starting point for that process in the discussion and coding examples provided.

Although the focus is on horseracing, the framework and the discussion of principles involved should make it easy to see how automated betting can be applied to horseracing in other countries, and indeed to other forms of sports betting. So let's continue by providing some further context on how complete automation of the betting process is now possible.

The IT revolution in horseracing betting

Much has been written about how the advent of betting exchanges has revolutionised the betting world. This is true in many ways that have become common knowledge to anyone with even a passing interest in betting – since exchanges are effectively a form of person to person betting, the odds available are truly determined by what the market is willing to offer. Now we can all act as bookmakers and punters, with the opportunity to lay as well as bet. The market is transparent, so you can win or lose as much money as exists in the market at the prices available – there is no such thing as a bookmakers limit or being "knocked back", you can get on for the prices and amounts shown. Last but not least, because the odds on offer are truly formed by a market (with the exchange taking a commission on winnings as opposed to formulating the odds itself) the prices are typically much better than those offered by fixed odds bookmakers, absent the margins built into bookmakers' overrounds on the sum probability of all horses winning the event.

One important aspect of online betting exchanges, (in particular Betfair, the world's largest betting exchange and the focus of examples provided in this book), is that they are primarily IT companies, with Betfair being one of the few success stories from the multitude of internet ventures that sprang up in the dotcom boom. How much betting exchanges owe to the general IT revolution is implicit in their existence. Without the widespread rise of the internet, broadband services, and advances in website and browser technology, there would be no possibility for a successful on-line betting market.

Since these markets are electronic and transparent, a number of other opportunities are opened up to any backer and layer of horses, over and above the advantages already listed, when exchanges are compared to traditional betting mediums. These opportunities are less immediately obvious, and require more work on behalf of the player, but can bring great advantage when part of an overall strategy, relating to betting automation. At the simplest level, betting automation on the exchanges involves placement and then execution of a bet without the need to be there or to interact with the exchange application. But automation on its own is of dubious benefit without being linked to a strategy. Typically, the automatic execution of bets results from the implementation of trading models, systems or rule-based selection methods which would be too difficult, if not impossible and/or tedious to execute manually, since they require constant and vigilant monitoring of market conditions.

The IT revolution in betting is not just limited to exchanges, of course - it is part of a deeper trend which affects all areas of betting. Not left behind in the new betting ring are of course the traditional bookmakers, who still account for the largest percentage of the horseracing betting market in terms of money wagered. All now offer a significant online betting presence, with the ability to bet at fixed odds, including multiple bets, and

to manage your account, alongside significant new media such as spread betting firms, offering the same facilities in the spread markets.

So much for the act of betting. The analysis of horseracing, however, has always been about much more than the execution of a bet. Indeed, as we mentioned earlier, the *raison d'etre* for automatic execution is often to implement the underlying analysis, where there is usually a model, system or set of rules in operation to determine what to bet on. Devising such systems relies upon the availability of appropriate data in the appropriate form. Nowadays, more information is available electronically than ever before – from all aspects of horseracing form to market prices - although as we will see, some forms are more useful than others for the purposes of automation.

In summary, the IT revolution is transforming betting and horseracing analysis, as it has the rest of daily life. Players commonly access information electronically, place bets electronically and use various applications to be more productive in their betting activities, from form databases to spreadsheets.

However, this book will take the process one stage further: to show how the player can take advantage of this revolution to automate the whole betting process from selection to execution.

The Quest for Automation

Like any other process that has been automated, "betting automation" is perhaps best thought of as describing a process which was once manual, and now is not – or at least does not have to be. Automation, or creating programs to perform previously manual tasks, is usually a natural extension to a successful manual process, rather than sitting in opposition to a manual process. As such, any betting strategy which relies on data and execution mechanisms which are available electronically can, in theory, be completely automated.

How much of the betting process is automated is of course down to the preferences, not to mention limitations, of the individual bettor.

We can make all parts of the betting process more efficient, or create useful utilities to help with certain tasks. In the case of creating a useful utility, such as for trading purposes, the objective is not to remove all user interaction for the task at hand, but to create useful applications which require, and indeed rely on, the user's interaction – although they may still speed up what was possible manually.

The perspective of *Automatic Exchange Betting* is to look at automating the entire betting process (which implicitly deals with the smaller as well as larger definitions of what is meant by automation) and supply examples for each stage, with the emphasis on making each stage work within the context of a complete strategy.

Whilst complete automation may not be the correct approach for every type of betting strategy (a strategy based on physical conformation as observed in the paddock, coupled with behaviour going to post springs to mind), it is surely the ultimate example of betting automation based upon a data driven approach.

Automating a strategy in this way not only saves time, but introduces an entirely new paradigm for betting.

A combination of speed of execution (e.g. obtaining prices before it is possible to do so manually), consistency of approach (so that a program can take a disciplined approach to betting on behalf of the bettor), and scalability (so that the computer can calculate and execute a number of bets which would be impossible to do manually) creates a level of reliability which is otherwise impossible and means hitherto impossible betting strategies become feasible.

.

Implementing an automated strategy also creates a different interaction with the activity of betting itself. A bettor can now be absent from the computer or bookie for an indefinite period of time, perhaps enjoying the spectacle of racing itself, or doing something more pressing. Of course, there is still interaction with an automated strategy, in order to review, modify and improve it, but this can happen on a timescale determined by the bettor - rather than by the next race, the next meeting, or even the next day.

PART 1: EXPLORING REQUIREMENTS FOR BETTING AUTOMATION

Chapter 1: Exploring the betting process

Here we explore the betting process in order to uncover the essential ingredients required for implementing a betting strategy. We will assess each of these elements in turn and look towards the key points to be gleaned as far as process and potential for automation are concerned.

The objective is to identify a repeatable path to automating profitable strategies, by defining a logical framework that can be used as a general reference for implementation of programs. The framework is therefore rooted in fundamental principles that apply to manual as well as automated betting.

"Traditional" betting and automated betting

We begin with the premise that any manual betting process that can be described can potentially be automated. The deciding factor is the extent to which each stage of the process can be quantified, or measured, and, in the case of data used to make that assessment, whether that data is electronically available for consumption by a computer.

The good news is that competitive, professional sport is generally a very quantitative business – both supporters and regulatory bodies tend to be obsessed with which contenders are the best, better than others, under what conditions, by what margins and according to what criteria. At a professional level, such measures are usually recorded wherever possible, thereby creating useful data.

In a traditional betting environment, all such data can form input to the betting decision making process, however it is stored, electronically or otherwise. Our job is to consider how closely best practice in finding winners and betting on them uses this data, and whether such approaches lend themselves towards automation.

Typically, traditional betting approaches, and by definition the majority of literature on betting in a pre-exchange world (where laying a contender to lose enters the equation), focus on the practice of picking and backing winners.

However, any bettor knows that finding winners alone does not, of course, guarantee any profit whatsoever. Since profit can only be produced if the returns from winners exceed stakes on the number of losers, it follows that any sound process must consider price relative to winning chance, as well as staking. This requires some assessment of what the winning chances are prior to betting, together with a staking plan.

Thus, the traditional, savvy bettor, is looking to identify the winner of any given event, and then only to back such winners that can deliver a long term return on investment - or in other words, in any event on which they are betting, they are looking to beat the market assessment of price. To beat the market, there must be a notion of fair price and therefore value about any potential bet. This is therefore the process we will attempt to mimic for our automated betting strategy.

Additionally, if we say that defining fair value is a feature of the traditional betting process, we can see that it simultaneously provides a viable approach for identifying laying opportunities, since backing one or more contenders in the same race is effectively the same as laying all the others.

In other words, backing a horse (or multiple horses), to win a race is effectively the same as betting everything else to lose, or laying everything else. This point, not so well understood before betting exchanges arrived on the scene, is now crystal clear in looking at the P&L position on any given race.

Trading is the only other possibility for an automated betting process that does not seem at first sight to be covered by reference to the traditional betting process. Although the possibility for active trading on every race did not exist before exchanges, a betting ring has always been subject to market forces of supply and demand, and looking for the best available price has always been part of the betting process, including an awareness of market forces lengthening or shortening the price. Bookmakers of course will hedge and trade with each other to offset their liabilities.

Aside from using knowledge of form to predict likely drifters or steamers, trading on a betting exchange has more in common with financial markets than traditional betting activities. Thus we will give the activity of trading a placeholder at this point – as an automated opportunity, it will be placed in the context of exploring market prices in the automated framework, since all trading relies on price and volume data – and explored in more detail in Part 2, in Chapter 8, in *Capturing and Using Market Information*, and also in Chapter 9, *Automating Betting Decisions*.

In the meantime, what key stages from the traditional betting process can we identify to take forward into our automatic betting framework?

Betting on one event

An event has a start, an end, and a definite outcome. It is clear a betting opportunity cannot exist without these parameters. The betting process focuses on one event at a time, with the objective being to identify the winner of that event.

It is true that a bettor might bet on multiples (i.e. on outcomes of more than one race, combined), but such decisions still rely upon the independence and integrity of the individual events that make up the multiple. Moreover, the bettor can only consider what chances any contender has within the context of the event in which it is competing, and then combine individual bets afterwards.

Considering the event type is also a first level of decision making that may apply to any bettor's strategy. We may choose to bet or ignore races on this basis; for example, ignoring Hunter Chases or amateur riders' races and considering all others, or ignoring all types except 2 year old Flat races. Such specialisation is often deemed to be a shrewd strategy.

Starting with the event details is a key point for us as far as automation is concerned, since it gives us the highest level entry point possible. Whatever else follows, we need to start with the event and consider one event at a time. The event conditions themselves are also our first decision point on deciding whether or not to bet. So retrieving the details of an event – whatever we subsequently define those to be - constitutes the beginning of the process.

Filtering by event type (e.g. To exclude types of races with a majority of contenders typically lacking previous form such as maiden races or national hunt flat races) on the exchanges is a simple matter.

Race descriptions are available data on Betfair, similarly selection of event type may occur in a bettor's own database, prior to interaction with the exchange API.

Simultaneous activity on multiple events is also feasible; this would simply be a case of running multiple processes. Events can be linked from a betting strategy perspective (e.g. Altering staking plans depending on recent profit or loss, or executing the automated equivalent of combination bets), which would involve providing additional inputs at a later stage of the automated process for bet execution. Such inputs vary depending on the overall betting strategy - from a framework point of view this does not alter our starting point of considering an individual event per betting process. We discuss automation in detail within the context of *Chapter 5, Betting Process 101: Daily Event Data*

Using Data to predict performance

We have stated that the traditional bettor is typically trying to identify the winner of the event, but the question that concerns us most in terms of automation is *how*. Of course, how this is done varies greatly according to the bettor's methods and is a huge subject in its own right. However, the good news from an automation point of view is that most attempts to find winners in a manual process involve using data either as an indicator of ability or likely future performance Therefore, we are interested in the extent to which profitable betting decisions can be made on the basis of data alone, whether these decisions can be described, and, if so, how they can be automated.

For the purposes of our discussion, we will first look at some of the most important common data elements that are used in an effort to indicate and compare winning chances, and then at the types of quantitative methods applied to those data elements to determine bets.

Form and Ratings

In the case of horseracing our standard unit of measurement is form. On one level, we have all the data attributes that appear on a racecard for any given event, from all the details pertaining to the race itself (e.g. race type, prize money, going, distance etc) and the attributes for each contender (e.g. weight carried on the day, stall number, jockey trainer etc through to the saddlecloth number) of each runner competing in it.

On another level, we can access the detailed historic performance (i.e. previous results) of every runner in the race, and how they fared in those races. The racecard usually lists data relating to raceday factors (although summaries of results, if only positions achieved, are usually included), with detailed results and analysis of previous races sitting separately to the racecard itself.

All of this data, once limited to specialist publications, is now available electronically, as we discuss in Chapter 3, and can be updated automatically as a data service. For our purposes, we are concerned that the data can be automatically accessed by programs which use the data to assess and bet selections. The objective in complete automation of the betting process is that there should be no manual intervention necessary. The challenge, however, for the manual as well as would-be automatic bettor, remains how the data should be used to do that.

Historic results can provide useful statistics not only about the runners themselves, of course, but also to build up a summary of other data elements which might be influential in predicting future outcomes – from performance statistics on trainers and jockeys, broken down by race type, course, time period etc, to statistics related to the percentage of winning favourites at different courses, broken down by race type. This type of information is particularly useful when historic form directly relating to the runner is absent, as in the case of debutantes.

Where contenders have a racing history, racecard form alone, in the form of summarising previous placings in races, gives us little to go on in terms of deciding the answer to the questions on which contenders are likely to fare best, better than others, or worst in any given event.

Nonetheless, basic attempts have been made to use raw form data from the racecard in the form of systems in order to answer these questions, including some of the most popular betting systems sold in the 1980s. One prime example involves assigning scores to the previous finishing position attributes available for each runner from the racecard. This simple scoring system can be exercised by any manual bettor with a racing card or newspaper since it takes as its input the previous 2 finishing positions of each contender (within the current racing season only), and also whether each contender has gained a course and/ or distance win. Points are awarded as follows: 5 points for finishing 1st (including a disqualified first), 4 points for second, 3 for 3rd, and 1 point for 4th place, scores being awarded for each of the previous 2 runs, provided that they occurred within the current season, with additional points awarded for any previous distance win (2 points) and any previous course win (1 point).

Applying such a model to any race gives us a set of scores for any contender between 0 and 13 points, where the highest rated might be considered a bet.

The model emulates what many traditional bettors have always done in analysing only those variables most common in the level of racecard form. By quantifying the approach in this way, we can see how it can be automated.

Defining a manual process, working out what are the most important elements in the manual process and whether they can be quantified are the fundamental prerequisites to automation. In this case, provided we have the data, such a method is easy to automate, but falls short of being useful or profitable. Amongst many issues, rating

finishing positions alone is usually comparing apples to oranges, since finishing positions may have been achieved in small or big fields, in different conditions, and above all in different racing classes.

It is interesting to note that using previous finishing positions alone looks basic now because racing websites such as *www.racingpost.co.uk* provide ready access to a wealth of information. Therefore most bettors have a more sophisticated view of what data variables should be considered – including a better way to compare the chances of contenders who have previously competed in different events, i.e. ratings.

Ratings methods establish a rating for each horse by assessing the merit of a horse's run in every race according to a known benchmark, typically against standard time or known competition within the race itself.

Unlike racecard data or a simple listing of previous finishing positions, one of the key advantages of using ratings for each horse is that they provide us with an absolute means of comparing one runner's ability with another. A further advantage of this approach is the potential to convert a rating into a probability of success, or odds, enabling us to compare an estimated price with the actual market, which we will explore in more detail later.

In the case of handicap ratings, the class of race and known form of horses who already have ratings will be considered, and other horses' ratings adjusted accordingly once a benchmark has been established. In the case of speed ratings, the time of the winner will be compared to benchmark standard times, and normalised for going (usually determined by the variation against other times on the day); similarly, once a benchmark has been set for the winning time, beaten horses will have their ratings adjusted according to distance beaten, translated into pounds.

The ratings are generally expressed in terms of weight, i.e. pounds, enabling ratings to be adjusted to a common scale for weight carried in handicaps. This follows the handicap ratings system or Official Ratings (OR) which are implemented and governed by the British Horseracing Authority. Measuring in terms of pounds enables us to allow for weight carried when assessing a new race, or to express superiority of one horse over another - so that we can for example refer to horse A being 10 pounds better than horse B. The official handicap scale runs between 0 – 140 for flat and 0 – 175 for jumps racing, so that Flat group races and Grade 1 jumps races would match the upper end of the scale in both. The assessment of the official handicapper is continuous, as more races unfold revealing further information about horses' abilities, such that a horse may have its rating continually revised without setting foot on a racecourse. The British Horseracing Authority runs a useful site which explains the handicapping system in more detail and also carries files showing the current official rating of each horse in training at *http://www.britishhorseracing.com*

However, since the official rating is set for the purpose of handicaps in order to equalise different abilities, the efficacy of using this rating to predict winners is questionable. The bettor is ideally looking for races in which there is demonstrable superiority and differentiation. Useful rankings are possible by using official ratings as a measure of ability in non-handicaps. Further analysis is possible by reviewing previous official ratings for the horses in question and how they performed against those ratings.

Since the greatest interest in identifying the winner is what advantage horses have outside the official handicap rating, some will compile their own or use private handicap ratings. There are a number of private handicapping rating services, one of the best used being Racing Post Ratings, widely publicised in the newspaper and on The Racing Post website. Ratings are produced for each race, as per the official handicapper, but unlike the official handicapper, they are also combined with a measure of interpretation in order to predict the likely rating that can be expected in a given forthcoming race, depending on the horse and the race conditions. Indeed, if compiling your own ratings, or using those of others, this approach highlights one of the greatest problems with which the bettor is faced: which rating to use for each contender in the current race?

In principal, the idea of using a single number which reflects a horse's likely performance in a forthcoming event is the perfect quantitative measure from which to base any manual or automated process. But of course, arriving at that number is not so simple, plus we need to consider how to account for ranges of performance. For private handicap ratings to be effective, it is argued that they must take into account a whole host of factors not immediately evident in simply benchmarking any performance against a known performer who ran to its form within the race. This requires a large degree of interpretation of each run – e.g. whether the horse was flattered or hampered by its draw, jockey tactics, the ground, the pace of the race; whether the race was up to the standard that might be expected within its grade etc. Also, using one rating to compile odds does not take into account any expected variation in performance, which is another important consideration.

Speed ratings are the other most common form of ratings used. Where speed ratings are compiled in the UK, the scale is often adjusted to reflect the same scale as used by the official handicapper, as with the Racing Post Topspeed ratings, or on a different scale but where each unit is still measured in terms of pounds, as with Timeform ratings. Essentially the objective is the same as with handicap ratings: to produce a rating which reflects the performance of every horse in every race which can then be adjusted to enable comparison between contenders for a forthcoming event.

With either a speed or handicap ratings approach, it should be clear there are many limitations regarding the type of race where they can be applied. Not least of the limitations of ratings as a universal approach exist where there is little (if any) previous form to go on, as is often the case with maiden and novice races, as well as early and late season races for younger horses at the start of their careers. For this reason, other attributes such as trainer trends (recent form, record in the type of race in question), sire form (suitability of sire's progeny to prevalent distance, going, maturity), and even movement in market prices as an indicator of "inside" knowledge, can be used to score potential ability as a number which can be translated to probability.

Even where ratings are relevant and sufficient previous form exists for meaningful comparisons, the most difficult question is often which rating from a horse's previous performance to use. For example, if a horse has run 40 times in its life, over the past 4 years, should we use the best rating, the best rating from the current season, the most recent rating, the best rating that relates to today's conditions (going and/or distance), and so on?

For this reason bettors may compile their own original ratings, based on detailed analysis of historic results (which may evolve into a completely automated process), to

take into account such questions in order to produce a rating that is representative of a horse's expected performance in today's contest. This is effectively representing a predictive model, in that it seeks to predict expected performance, which we refer to again in the section on *Assessing Chances.*

Whether compiling originally or re-using the handicap or speed ratings of others, it can be strongly argued that representing performance as a rating is an effective shorthand of form study - particularly when the rating is taking into account the factors that would normally be judged in any manual study.

If we argue that a quantitative rating method embodies an analysis of collateral form, then it would also follow that using ratings is a way of automating the manual study of form, which is an integral part of the traditional betting process.

Looking for Winners

To realise the potential value of form and ratings requires a method to identify winners or assess relative chances within a race, either for manual or automated betting.

At this point, many bettors argue that their own method for using form attributes or ratings in order to assess contenders cannot be defined in a systematic way, and varies race by race, depending upon their own interpretation of raw data – therefore defying attempts at automation.

There are equally many attempts in a traditional betting process to do precisely the opposite – to apply methods to data to systematically create profitable strategies. There are two sides to this coin – one is looking for winners and one is assessing chances. They are both related, but the emphasis is different. We will look first at the systems approach and in the next section consider approaches that do not attempt to identify the winner of the race alone, but rather use the same data to assess the relative chance of all the contenders, where price is key.

Betting systems

Although definitions can vary, the typical objective of a betting system is to identify horses to bet by applying a predetermined set of rules, based upon previous analysis of form variables and ratings, or to pass on the race if there are no candidates to which the rules apply.

The motivation is to rapidly – even automatically - identify a system bet by applying the rules, rather than evaluating all contenders' relative chances. In this sense, using a betting system alone does not provide a generic approach to identifying the winner of any given race, and can only be seen as one tool in a bettor's kitbag rather than as best practice.

On the other hand, a betting system could be to back the top rated speed horse in a race. If that is the case, we would be combining an existing method of ranking all the contenders relative to an absolute benchmark, and a system for identifying the winner.

Systems can therefore be a short cut to identifying a bet as quickly as possible, following easy to interpret rules, as opposed to the lengthy time spent in front of the form that may be required to manually weigh up the chances of all runners. The appeal is summed up neatly (and it should be added with some irony, since the author advocates using systems as an educational tool and taking a dynamic approach to devising systems) by the title of Nick Mordin's book on the subject, *Winning Without Thinking*.

Indeed, profitable betting systems will often start by identifying some aspect of form or consistent variable in results that can be measured (such as the statistically significant winning strike rate of favourites) or looking at unusual changes in circumstance (e.g. wearing blinkers for the first time, changing trainers etc.) which are then combined with various qualifying factors, such as race attributes (e.g. Maiden and/or handicap) or horse attributes (gender, age etc). The principal in any profitable betting system is that the significance of the system's factors is overlooked and therefore undervalued when other bettors are making selections. Lack of betting support leads to longer market prices than the systematic chance of winning has predicted, meaning that if bet over the long term, a profit will be made equivalent to the difference in the two (provided of course that the market always significantly undervalues such factors as are exposed by the betting system).

From the automatic betting point of view, it may also seem at first sight as if any betting system that has shown a past profit, with easy to implement, unambiguous rules provides an ideal candidate for automation (so that bets are never again missed, rules are consistently applied, multiple systems can be implemented at once, time can be saved etc.)

However, to coin a phrase, showing a past profit for a system is no guarantee of future success. At a minimum it is useful to see the profit made (not always forthcoming in some third party systems), the length of time over which the system was tested, the number of qualifying bets, the average price and the strike rate.

If the system has stood up to real tests in the field, all the better. When devising systems it is easy to succumb to "overfitting", particularly where profitability is the main criteria for success. Thus, any betting system's results can be skewed by the presence of one or more very large prices within the qualifying data set. Since the prime motivation of the analyst is to find a profitable system, there is a constant danger that he will devise a system that only fits the historic data and has no relevance for the future.

Clear danger signs are limited data sets and convoluted rules that seem to have no logical basis (e.g. back the favourite in all 2 yr old races at Salisbury if the last form figure is a 3 and the trainer is based within 30 miles of the course). The corollary is also true – if the reasons for success of the system are evident in the rules used, then we may have the basis for a profitable system. For example, betting all contenders over a certain dosage index score as an indicator of stamina, to identify potential Derby winners and St Leger winners where most race contenders are previously untried at the distance, contains logic within the rules.

There is also the question of how long a system may remain effective. Even if the profits, strike rate and average price yielded by a betting system are impressive, any edge in price produced by a winning method relies on the market remaining unaware of

that edge. Thus systems are transitory insofar as a market is not constant and can adapt to flatten out profitable angles.

Use of speed figures is a good case in point. When the Racing Post first published the Topspeed ratings in the 1990s, they were not widely used as a means of predicting winners in the UK, despite being a good indicator of ability and therefore performance. Systems based around speed figures therefore had a profitable edge. For example, blindly betting the contender with the best overall Topspeed figure in a race where the contender had also attained the best last time out Topspeed produced a healthy profit. The strike rate on such selections remains more or less consistent, showing the raw power of the rating, but over time as more and more bettors incorporate these variables the market opportunity, and therefore profitability, disappears. So, although the number of winners thrown up by a reliable system will be a constant, the price of winners is always what gives a system its edge – or not. Hence the "contrarian investing" elements of betting systems, where developers of systems look for significant angles that are ignored or underestimated by the crowd (or market as a whole), leading to overpriced selections.

Assuming for a moment that a bettor has identified significant systems – i.e. the system is effective in making a profit, and the caveats applying to the system have been reviewed - automation is a very effective method of execution.

Any bettor who has attempted to follow systems manually is aware of the sheer tedium and room for error in browsing daily racecards and form in order to apply any given system. Even where a form database is used by the bettor, following a betting system can still be very inefficient without complete automation. Depending on the form database, and the way it is set up, searches may be interactive, the latest results or racecards may be missing, or it may be necessary to check results or apply rules daily, and so on. This in itself can be an inhibitor to following a system and reaping profits, and is an obstacle removed by complete automation.

The benefits of automation for betting systems are several and tie in with general themes discussed earlier. Since any system requires rigid adherence in order to produce the same results as a test run, automation enables selections to be generated automatically, freeing up the time of the bettor and ensuring errors or omissions are not made. Likewise, time will be saved and errors avoided in betting the selection(s).

Often the bugbear of followers of betting systems is that they have missed betting a selection due to other commitments, or that it is not feasible to follow a system in the first place because of the manual intensity of the process and the time required. If starting to follow a system, there is a constant tension in the case of missing the winners – especially those at big prices - which rely on a system making its returns, particularly in the case of those systems with low strike rates.

Given that the betting system has already been developed, there are three parts to the process of automation that are generic – automating the selection process for the betting system, specifying bet conditions and executing bets according to those conditions.

The feasibility of generating selection system picks is the same for simple or complex criteria. Provided that the data upon which selections are based is readily available and well structured then there should be no issue. Development of a system is a

different question, and will require at least some interactivity on the part of the developer Once a system has been developed there is preparatory work to construct and set up the correct queries to run against the data on a daily basis (and possibly more work for complex systems).

The required output from any system must identify the runner in question (by which we include the event and any other parameters necessary for execution on Betfair) and specify some betting conditions, so that it can be bet automatically. The question of data sources and accessibility is covered in Chapter 3. We discuss the requirements for implementation together with examples fully in Part 2.

Missing from the discussion for betting systems so far is any reference to building in a minimum price requirement. Expected return on any system is key to profitability. Ways to quantify would include looking at strike rate of the system when tested over a reasonable period of time or applying any minimum price filters that have been applied in the selection rules.

In a manual process, obtaining minimum price on a system selection is often ignored as a requirement. However, it is a fair assumption that most system bettors if presented with the prospect of an extreme scenario (e.g. their selection picks are only available at 1/100) would of course decide that the bet was “not worth it”, in other words that it is below a minimum acceptable price. If we imagine offering bigger and bigger prices about a selection in which there is no interest, as a layer would, we can imagine there will be a point at which the system bettor would deem a bet as being great value – the question for any system is what is that point and how do we determine it?

Given that relative probability of winning, and therefore price, is so pivotal to profitability, it makes sense to consider it explicitly and to look at best practice for doing so, so that it can be applied within the automatic betting process. A discussion of assessing chances enables us to progress to that.

Third party sources

Standing outside the previous discussion whereby the bettor is employing his own methods to identify winners is the reliance of many bettors in a traditional setting on third party advice or ratings – these also can be viable candidates for automated strategies.

There is no shortage of information sites, ratings and tipping services dedicated to racing and other sports now on the internet. An automatic bettor has the choice of deciding whether to use any such information where they have found it to be valid or useable “as is”, or to use their own methods to adapt it, or a combination of both.

Whilst following independent advice curtails many stages in the betting process that are being considered (effectively outsourcing responsibility for such stages), it is perfectly well suited to an automated approach, provided that the advice can be represented as an electronic input, such as a text file containing relevant bet details that can be collected and manipulated automatically.

At a minimum, if using third party advice or ratings, information relating to race, time, contender, and price is again required. If the data is a rating or odds, the bettor's own program can apply further logic to convert the ratings to acceptable odds, or to apply other rules in terms of minimum acceptable price. In Chapter 3 we discuss screen scraping, use of APIs and so on which can enable retrieval of this type of information, assuming it is electronically available.

From an automation point of view, programs will need to be scheduled to retrieve the data, then set up further tasks to apply betting decision rules and execute bets as appropriate, matching the selections with the Betfair identifiers for market and selection Id, as required.

We cover these processes in Part 2, with particular reference to Chapter 7, *Scheduling – The Key To Automation*, and execution of bets from any input file in Chapter 10, *Betting Execution.*

Thus, there is no reason why third party selections or recommendations at certain prices cannot be automated; this would effectively replace the stage of selection by the bettor.

Observations prior to an event

Bettors who attend the races have a unique opportunity to assess the chances of contenders according to their appearance or behaviour prior to the race. Typically, the key stages are behaviour and appearance in the pre-parade ring, in the paddock, behaviour on jockey mounting, behaviour and appearance going to post and entering the stalls, or prior to the off.

Information from these observations can have a significant bearing on performance, from horses that look particularly well, move to post well and have a relaxed demeanour to those that look unfit, move to post badly, or have reacted badly to some aspect of racing preliminaries. None of this influential behaviour will typically be taken directly into account in an automated betting system, ratings method or predictive model, since the variables used are based on historic data.

Additionally, last minute changes in going, weather (including factors such as wind speed), draw bias, which can have a profound effect on results, are typically not taken into account in methods which rely on automation, and can be more difficult to pick up in the automated betting process. Some of these factors can be quantified. For example, draw bias at tracks can be pronounced and particular to multiple day meetings depending on the ground preparation for that meeting – so it is possible to assess that from the first few races; likewise with going.

The ability to take into account last minute observations before events is a good argument for saying that the human brain is the ultimate decision making tool for all bets. Indeed, this should be the case in terms of being able to take into account all possible factors and come to a conclusion based on the latest evidence. However, there is no guarantee in the case of last minute information that all factors, including historic

performance, will be equally balanced in the decision making process, or that any decision would be consistent given all similar inputs in a future case.

In some cases, careful balancing of decisions and inputs is not needed and it will be obvious to on-course bettors that certain horses are disadvantaged by new information relating to their chances (e.g. the horse bolts before the off, expends great energy yet is still allowed to compete), so that information can occasionally be missed, if up to the minute observations are not part of an automated strategy.

On the other hand, as with many of the arguments over whether an automated strategy can be effective, the question is not whether the strategy is without flaw, but whether the edge gained from the strategy outweighs the impact of missing certain inputs.

Notwithstanding the above, there is information that is available to the automatic bettor that can take account of changing variables. The possibility always exists to inform the automated program, or a number of automated programs, via reports from the track, although in the case of paddock and going to post observations, this information may not be readily available and can be very subjective. More reliable is information on such variables as intra day going, which can be picked up in an intraday results service, and is available in a format on a number of websites that can be picked up live.

Assessing Winning Chances

If we use betting systems to pick potential winners, we get a selection but have no idea of its relative chance within the race in question, only the past performance or profitability of previous qualifiers for the system over many different races.

The greatest limitation of this type of approach is that it does not account for the relative strength of the competition in the race, and therefore ignores the probability of the selection's winning chance compared to its closest rival, or the next ranked rival to the one below that, and so on. The impact this has upon the bettor's ability to make a profitable betting decision is significant, since any concept of a "value price" is based upon the run of previous systems qualifiers, rather than the competition within the race. Similarly, an approach that only looks for the winner limits the bettor in taking advantage of other betting opportunities – such as dutching, backing to variable stakes and laying - where an assessment of each contender's relative chance, or probability of winning, is crucial.

Predictive models

We use the term "predictive models" somewhat broadly here, mainly as a contrast to betting systems, to cover any model that attempts to predict each horse's likely performance in a given race.

The key differentiator, as we discussed in the section on ratings, is that the ideal output from such a model will enable us to compare each contender in an upcoming race according to a numeric value – this can be a rating which can be converted to a probability, or an estimate of the probability itself, between 0 – no chance, and 1 - the winner.

Unlike speed and handicap ratings, the objective is not simply to provide a scale by which we can measure a horse's performance in any given race, past or present, or to compare a horse's performance from one run to the next.

Rather, the objective is to define a model which determines the relative chance of winning the race in question, so that we can assess the range of difference between each contender in that race – and thereafter estimate the probability of winning. The two types of rating are related, but differ in purpose.

In the case of building a predictive model, there are many variables which we have not yet considered that may be relevant, in particular suitability of the contender to the race in question. This may include suitability of the contender to the going on the day; suitability to distance; suitability to course; the class or ability of the animal – that is whether it is up to the standard generally required by the type of race - and so on. We can score attributes on a binary or categorical basis (i.e. yes or no, 1 or 0) or on a sliding scale. Whether we choose to do so will depend to some extent on how we want to weight or attach relative importance to the different variables in determining the overall score, and also on the characteristics of the variable itself (e.g. Suitability to going is not

usually binary, so that a horse that acts well on good ground may also run well on good to soft or good to firm)

Furthermore, there is no reason why we cannot adjust the weighting given to different variables. This can be done manually (e.g. We might experiment with the value of one variable, such as trainer form, and see if increasing this value has more predictive power) or by using predictive modelling techniques with the aid of computer software.

The purpose of using formal predictive modelling techniques, such as statistical modelling methods, like linear regression, or machine learning methods, such as neural networks, is to help us determine what influence specific variables and combinations of variables have on performance, with what “weighting” they should be applied, and therefore what “scores” or probabilities should be produced as the output.

In the case of statistical modelling, the creation of the model is in the hands of the statistician or analyst, who will use statistical methods to explore the data, which must include the variables and results, in order to assess which variables or combination thereof have the greatest predictive influence. From there, a model can be created that can take future variables of the same type and range in order to forecast probabilities for each contender based on that model. Of course, it may well be appropriate to create different models for different types of races, and certain inputs may be more relevant for some race types than others.

In the case of machine learning and data mining methods, the principal is the same except the emphasis is on automatic or semi-automatic generation of the models. This means that the prediction process can be treated as more of a “black box” that will “just work”.

Datasets will consist of a row per horse, in the form of a series of numeric values relating to each variable to be used, and a desired output relating to the result of the race, either in binary format (for won or lost) or some form of relative performance (the placing or lengths beaten for example). Data preparation will still be done by the developer, but applying modelling techniques such as neural networks to correctly formatted data, also known as “training” the model, will create a model automatically. Some understanding and experimentation will still be needed. Subsequently, it is considered best practice to save some of the dataset to test the model which has been created, to ensure similar accuracy in identifying the winner or assessing probabilities has been attained.

Thereafter the model can be applied to new races.

Later, in Chapter 6, *Assessing Contenders*, we will look at automatically repurposing the available data in the Racing Post’s Postdata service, which broadly measures ability and suitability of each contender to prevailing conditions on the day, for the purposes of creating a simple example to predict winning chances in the form of an oddsline. In Part 3, we also introduce a set of third party ratings, which apply a predictive model based on handicapping methods, and compare the two approaches.

Defining value and using oddslines

Following the logic of assessing chances in the betting process comes the concept of defining value by comparing the "true price" for each selection with the market price.

The archetypal example for explaining the concept of value is to use the example of the toss of a coin, where we know the true probability. Barring the possibility of deformities in the coin, and given a fair throw, we can agree that the probability of a coin landing on heads or tails is 50/50. In other words, there is an even probability on either outcome. Therefore, accepting odds offered by any layer on such a toss of less than evens, or in other words less than 50/50 on one of the outcomes occurring, would mean a bad value bet.

The bettor may still be prepared to bet at less than evens (at this point he is more of a gambler), and may also win, but we know that when he is consistently betting at a return which is less than the probability of the event itself occurring, then the layer will win in the long run, and the backer will lose in the long run. The key is that given enough trials, we will expect to see the distribution of heads and tails converge towards the "true" probability of the event occurring.

Similarly, if the layer were offering odds on heads or tails at a price greater than evens, say 2/1, we would expect that is a good value bet. The backer may lose the next toss he bets on, (there is still a 50/50 chance that he will lose, after all, despite the attractive odds) but if the return on every successful bet in these circumstances is 2/1, it should not be long before the layer is broke and the bettor has taken all his money. The key factor again is that if there are enough trials, and the bettor leaves enough money to back on each trial, there is inevitability to making or losing money in this way.

So far, so good - as far as the concept of value is concerned. The problem – or perhaps the opportunity - with horseracing is that no-one can say with certainty what the probability is that any given contender will win. But we can define some boundaries, and having done so can start to move closer to a solution. To go again from the toss of a coin, we can say that if all contenders have an equal chance, in other words if we assume that horseracing is just a random game, then the probability of any contender winning is 1/Size of Field, or in terms of fractional odds, it is (Size of Field-1)/1.

Therefore, if we are convinced that any contender has a better chance than random (i.e. better than the probability determined by *N* runners in the race), and the odds being offered are (Size of Field-1)/1, then we have found a value bet.

By the same token, this means that - given the probability of each contender winning must add up to 1 in the event – in this event there must also be at least one horse in the field with a worse chance than the random chance. Which horse or horses are they? What about the other runners in the field, do any of them have a better probability than the reciprocal of the number of runners in the race?

In an iterative manual process, we can imagine poring through all the runners, adjusting and readjusting our view on each of their chances in order to determine this. At each pass, we would be creating a new oddsline.

An oddsline therefore tends to provide a summary view of what the real chances are of any contender in a race. Examples of oddsline outputs are repeated through Parts 2 and 3, in particular Chapter 6, *Assessing Contenders*. We can use an oddsline as a map in order to determine what a value price is when we go to the market. Here, we are concerned with the ability to determine value automatically.

We have seen from all methods of assessing chances of winning that each method implicitly produces a price or set of prices which can be compared to the actual market.

Sometimes the bettor's notion of prices for all runners can be vague, depending on how quantitative is the method of assessment, but it is still present – for example, in the bettor's mind it can be an assessment of price which applies only to one runner, in the case of establishing a minimum price for a betting system pick. Once present, the remaining probabilities in the race belong to all the other runners, thus, even when we start with a vague idea, there is still a concept of prices for all. With methods where the analysis determines a score or explicit probability for every runner in the race, producing a set of prices for all runners is a logical outcome of applying the method.

As we discussed at the beginning of the exploration of the betting process, identifying winners is not enough to ensure profitability. Since the savvy bettor is looking to identify winning chances in any given event, but only to back such winners that can deliver a long term return on investment, there must be a notion of fair price and therefore value about any potential bet. Therefore, a set of prices based on probabilities for all contenders summing up to 1 makes for a bettor's oddsline which can serve as a map to use to navigate the actual market. We can convert these probabilities to odds, in order to use them as a set of "tissue prices" with which to compare to the actual market.

For automating the entire strategy, we can generate an oddsline automatically using predictive modelling techniques, either devised by the bettor or using third party sources, or converting a set of ratings or scores to odds. In order to arrive at the most accurate prices possible in the oddsline, we can assist the process by backtesting actual results from races against our predictions, then making adjustments in the method used. We look further at this process after running live oddsline strategies in Part 3.

Generating an automated oddsline is therefore a useful starting point for our implementation examples – since using betting systems, and other strategies can be seen as a subset of the implementation tasks that have to be set up for any betting strategy using an oddsline.

We use the oddsline as the basis of our example automated strategy for the framework in Part 2, and run live automated betting strategies based upon oddslines in Part 3.

Using Market Information

Traditionally, using market information in the betting process has been a matter of assessing which prices represent the best value about any particular selection – which as we have seen requires an assessment of a fair or "true" price about contenders first. Or, having decided to bet, analysing market information was about obtaining the best offer available on our selection.

Until the advent of exchanges, the betting ring or on-course market (shaped in large part by the off-course bookmakers), was the only market worthy of consideration in terms of real liquidity, barring the most popular races where the ante-post or morning markets would also play a part.

Bettors in this market, whilst having the possibility of dutching, could not lay, and had to go to multiple bookmakers in order to find best price. Comparing prices to the bettor's oddsline or assessment of value and "getting on" was a frenetic activity.

In the age of the exchange, the market information available has in many ways moved closer to that of financial markets than the betting ring from which it has evolved, opening up many more money-making opportunities to bettors in addition to consolidating those mentioned.

Three parameters on the exchanges are of critical importance to inform the decisions of the bettor, mirrored in real time and historically. In real time, we have the back and lay price offered to various depths of price, the volume available to the various prices, and the timing, or time at which those offers are available.

Additionally, we have the traded market history in terms of all matched (rather than simply offered) bets, the prices and volumes at which they were matched, and the time at which the bets were matched.

These parameters enable us to assess the liquidity and competitiveness of the overall market, which again is an input to any betting decisions, as well as trends and momentum in the market, and a fair overround (i.e. the most advantageous set of prices to bet into).

The advent of exchanges and a proliferation of online betting sites, both for assessing latest bookmakers' odds and betting online, have opened up other opportunities to benefit from trading or hedging on price alone, including arbitrage opportunities between all odds providers.

However, from an automated betting perspective, whilst obtaining prices is easy enough, ensuring we can get on, or execute bets with multiple odds providers is not. Arbitrage between differing providers of odds falls outside our definition of backing, laying and trading on the exchange, since there are many challenges – not least that fixed odds providers can refuse bets, and automation via fixed odds providers' sites, whilst possible, is not reliable or supported. To reliably automate capture and use of market information, we therefore focus on using the exchange API.

The bettor intending to back or lay one or more runners in a race will typically compare his assessment of winning chances for each contender, or oddsline, with market information, before taking a betting decision, as before the advent of exchanges.

However, since markets are now dynamic, he can realistically decide whether to use market information to help decide where the market will be, not just where it is now, and speculate on obtaining a price not yet available, specifying a conditional price at SP, and so on (options regarding betting execution that we leave to that stage of the betting process).

For the trader, capturing and using market information can be considered the start of the betting process, if we are looking to make a profit based on movement in price, and lock that in before the race is run. So there may be multiple visits to the market to achieve this.

Of course, there are many shades of grey in what can be done with market information, combining the approaches of backing, laying and trading. This can include hybrid strategies which may centre around trading a particular contender or set of contenders' prices which are deemed value by the oddsline, but where price fluctuation allows locking in a profit as well as the possibility of achieving a "bet to nothing".

If the bettor can find a method of assessing contenders' fundamental chances which reliably predict price movement, then betting and trading lock together. Similarly, the presence of in-running markets and bet persistence allows other "back to lay" or "lay to back" strategies that can be considered a hybrid between trading a price and betting on an outcome.

We discuss programs to enable a range of betting and trading strategies in Chapter 8, *Capturing and Using Market Information.*

Deciding whether and what to Bet

A qualitative betting decision can be taken at any time for any reason. However, in terms of describing a quantitative betting process, the decision on whether to bet is in logical sequence to the rest of the process we have described, relying on previous stages being completed.

Typically this will mean the bettor comes to this decision point having assessed contenders in the race, resulting in a view of probable performance, and having reviewed the market. In the betting decision making process, the bettor combines these inputs and determines whether or not to bet, what types of bet to place, and how much to stake according to his betting strategy. This may be laying, backing, dutching, or the strategy itself may be determined by the inputs received – based on where the market offers the greatest value or opportunities. Although the stake is also determined at this point in the process, we will comment further on the subject of staking within the next subsection.

Therefore, the betting decision may be simple or complex, depending upon the betting strategy itself.

Where we are working from an oddsline, the first step is to compare the selections in the oddsline with market prices. However, determining whether to bet, how much to bet and the selections to be bet will be triggered by the rules of the betting strategy itself.

Theoretically, any price over the oddsline's assessment of probability is good value; any price under the assessment is bad value – at least in a perfect oddsline. Since the perfect oddsline does not exist (some information is always unknown, even assuming our method is flawless), the objective is to beat the market with any edge gained in the method. There are many decision points in this process that lend themselves to a multitude of different strategies. Some of the factors to be considered include:

- Should the comparison only consider a limited number of runners in the oddsline? Or for what proportion of the field should we apply it (e.g. Top quarter)?
- What absolute difference or proportional difference in price must exist in order to determine a value selection and therefore a bet?
- For value selections, is this a backing or laying strategy, or both?
- Do we want to go with or against the oddsline? In other words, do we trust our oddsline well enough to lay the top rated horse from our oddsline, even if it is rated as having the best chance, if the market price is lower than the price or probability we give the contender?
- What are the maximum odds in the oddsline at which we are prepared to play? For example, if backing we may want to concentrate only on the upper part of the market to avoid the possibility of punitive losing runs (To take an extreme example, can the bettor really be confident in a strategy which has more than a 50% + chance of a losing run of 100 + before a return?), even if there is value identified in longer priced selections.

Many of these questions will be determined by how confident the bettor is in his assessment of probability, and therefore how far they want to push reliance upon the oddsline.

Testing strategies in advance to determine what has proved profitable in the past can therefore help greatly with these decisions, as we demonstrate Part 3 of the book, *Automated Strategies in Practice.*

From the perspective of automation, once we have defined the logic of the betting strategy, as in the examples above, it is a matter of creating the program to take the inputs and apply the decision-making process. This stage can be the creative part of the programming process, where the bettor's own thinking, or strategies based on the bettor's research, will be implemented within the automated strategy. For the examples described, such as comparative analysis of oddslines versus the market, as well as applying various filters, the process is relatively straightforward.

For more complex strategies which rely on market movements to generate profits such as hybrid trading strategies, deciding whether and what to bet can be an iterative process taking more inputs, such as recorded bet details, revisiting the market and applying logic on staking and execution in order to lock in profits.

Implementation examples and discussion within the context of the automated framework are provided in Chapter 9, *Automating Betting Decisions*.

Staking

One of our earliest statements in exploring the betting process was that picking winners is not a profitable activity unless the collective price obtained on winners bet outstrips the number of losers bet By the same token, it doesn't matter how many winners are picked if staking is inconsistent - which is often the case in a manual process. So automating staking in order to produce consistency can really help profitability – especially so for any bettor whose staking is inconsistent but whose methods are sound and can also be automated.

The simplest - and often soundest - staking plan is to bet level stakes, in other words placing the same amount on each bet. That will minimize the effects of erratic staking, so that any method which produces a unit profit will in fact produce an actual profit. To begin, even for level stakes, we have to determine the size of the unit stake. A practical way to do this is to predict the longest losing run in unit stake terms that we might expect from our betting strategy, and ensure that the bank can endure this, with a margin of safety built in on top. Alternatively, during backtesting, we can assess the longest unprofitable period and ensure the bank can withstand this.

Thus we might begin with a bank of 100 points. To ensure that the bank grows with profitability of bets placed, we can go further, and divide the bank by the number of units appropriate to our strategy at some regular interval – every bet, every 10 bets, or at a time interval.

More complex staking plans are designed to increase returns beyond level stakes depending on the edge of the bettor over the market. To operate such methods quantitatively requires a more detailed analysis of expected return or a very accurate assessment of chance in order to work properly, e.g. the precise edge given in the oddsline versus the probability predicted by the actual market. In the case of operating complex staking plans within a manual process, the bettor is effectively increasing the difficulty of their task, since not only does he have to find a profitable method, but, for a staking plan to work well, he has to determine the exact edge in the method prior to betting. Again, automation brings significant benefits if operating such methods since calculations can be implemented automatically in milliseconds prior to betting. As we discuss in our automated framework, an API implementation can retrieve the current Betfair balance to use as the bank, which is an important variable in any such calculation.

Provided that the bettor has confidence in the edge over the market for any given contender in any given event, the so-called “Kelly Criterion”, named after the methods devised by John L Kelly, is generally recognized as an answer to the problem of what is the optimal staking strategy in order to make the bankroll grow over the long term. The answer is fairly simple in principal, although the exact implementation can be more complex.

Kelly showed that the proportion of bank to be wagered on any particular event in order to optimise your bank's growth rate is edge/odds. By way of an example, suppose we have identified a 5-1 shot as being a good bet. In this context, for it to be a 'good' bet, we must believe that the 5-1 price available is generous. Let's say we believe that a price of 3-1 is a better reflection of it's true winning chances.

The 3-1 implies a win probability of 1 in 4, or 0.25
This means that the edge is .25 x 5 - (1 - .25) = .50
…and any wager should be 0.5/5 = 0.1 or 10% of the betting bank.

Estimating win probabilities is an extremely difficult task however, and many serious bettors prefer to wager somewhat less than the theoretical optimal proportions of bank - many opting for 'half-Kelly'. This would mean wagering 5% of bank in the above example.

Thus, using the Kelly Criterion with an oddsline strategy would dictate the stake to be used, determined by the implied edge between “true chance” and the market price. The maths for implementing “Kelly” over multiple selections means that on occasions a horse with no individual implied edge on the oddsline may be backed. John Haigh’s *Taking Chances*, covers a fuller explanation of Kelly and an exact implementation for Kelly in the instance of backing multiple contenders with an edge, although in the absence of a perfect oddsline, Kelly should be treated with caution.

Aside from level stakes, betting a progressive percentage of the bank to level stakes and a Kelly strategy, more complicated staking plans should also be treated with caution. Like betting systems, there is no shortage of methods employing quackery – the promise being to take an average or even poor betting system and turn it into a profitable one through application of a staking plan. Nowhere is this worse than in plans that advise increasing stakes by some proportion in the event of losing bets in order to recoup previous losses, whilst maintaining the same profit target.

Of course, any staking plan can be made to work with the benefit of foresight, so that if we knew which bets were losers and winners in advance, a staking system could be devised to compensate for losses, but this is the same as overfitting a betting system – it will only work on a specific dataset. In any case, if the bettor knew which horses would lose and which would win in what sequence, there would hardly be a need for a staking plan to ensure profitability!

Placing Bets

Arriving at a point where we are ready to place a bet is a culmination of the betting process but not an end to the decisions that have to be taken in order to "get on".

In the traditional betting process, the bettor will obtain a price at the time the bet is struck, whether this is simply to accept SP or take fixed odds (through the live market or an early price, or even to negotiate their preferred price in the ring). The bookmaker is the counterparty to the bet, and the bet is either agreed or not. If the bettor wants to hold out for a better price, he can wait for the market to move in his favour, but must then engage in the process again.

In the exchange, on the other hand, many options for execution now exist, split between the Exchange and Betfair SP (BSP) markets. In Exchange betting, the counterparty to the bet is unknown and offers can be placed and accepted, and left pending agreement. In this case, the bettor will only know if their bets have been matched or partially matched if they revisit the market. Although there is now the option to let bets persist through to the in-running markets, the same principal applies if wanting to ensure what has happened. Alternatively, a bettor can now place bets at a Betfair SP, or conditional BSP depending upon a minimum price, where the Exchange is effectively acting as the counterparty to the bet in the period between the bet being placed and a reconciliation of all bets placed. Although in this case there is no way of determining either the price obtained in advance, or knowing if the bet has been accepted prior to the off.

In the world of the exchange, it is not far to go from attempting to bet selections at the highest (if backing) or lowest (if laying) prices and revisiting the market to verify what has or has not been matched, to trading. We simply require an opposite bet, provided the market has moved in our favour, in order to lock in a profit at some later point.

Depending on whether we are attempting to get matched at the current market or go for a better price (either to back or to lay), the automated execution process will be more simple or complex.

The first stage of execution is to place a bet, which requires one call to the API, provided event, selection name, price and other parameters relating to the bet requirements (back or lay, stake etc) are known.

If going for a match at market price within currently available to bet volumes, there is an excellent chance by using the API that the bet will be matched at once – provided that the price has been retrieved immediately beforehand. In such cases the automatic bettor may be content to leave the execution process at that and run a program at the end of the day to report all matched bets.

On the other hand, if there is a requirement to place high volumes or amounts at prices outside the current market range, it will certainly be necessary to enquire again on the bet and market prices, and may be necessary to engage in a second stage of execution. This will determine whether or not the bet has been matched, and if so how much of the bet is still outstanding. A reassessment of the market will then be required, to establish

new parameters for executing the bet, referring back to the strategy logic attempting to match the remaining amounts.

The good news as far as automation is concerned is that all such operations are possible, and more, given that we are dealing entirely within the Betfair API at this stage. Efficiency of execution will all come down to our automated implementation and the logistics of programming each strategy correctly. We cover this in detail in Part 2.

Record Keeping

We include record keeping as a final step in our exploration of the betting process. If we view betting as one off process, then recording and analysing bets is of no consequence. However, we have taken the view that if the process is to be repeated, then the objective is to make profits in the long term. The importance of recording bets is also referred to in the best literature on the subject as well as by successful bettors. Not only is it critical to keep an eye on profit and loss, but also to refine and improve strategies by understanding successes and failures within a given strategy.

For our automated process, record keeping takes on an additional dimension of recording the betting process itself, since the bettor - in a traditional sense - is absent! This includes program output to record the process of generating selections right through to the prices obtained, an audit trail for betting decisions made, and recording any bets placed. Equally important in this process is trapping any errors which may occur. Thus the automatic bettor is interested in improving execution of the process as well as the strategy behind the bets.

One of the main benefits in automating record keeping as opposed to manual tracking comes from the ability to capture and store many characteristics of an individual bet. This includes the standard bet details from the date, time, event type (code, handicap, and distance), conditions on the day (e.g. Going, number of runners, draw), as well as the elements that led to the bet in the first place (for example, top rated horse, percentage overlay etc), so that the bettor can analyse their betting by slicing and dicing these factors in the future. Furthermore, if different systems or strategies are being operated, then the bettor can set up programs to track how well each system or strategy is doing separately from the main betting record, adding further dimensions for analysis. By reviewing such information, the bettor can optimise their strategy over the long term to focus on strengths and eliminate weaknesses.

Chapter 2 : The Automatic Betting Process

We explored the essential ingredients of the betting process in Chapter 1 and the process by which each of these elements could operate together within an automated framework.

Betting on one event, selected from a universe of events:

- Identifying opportunities within the event and assessing contenders' chances, applying:
  - Betting Systems
  - Ratings Methods
  - Predictive Models
  - Third party advice and ratings
- Determining "value price" and creating oddslines
- Capturing Market Information
- Betting Decision
- Execute Bets

We can represent the way in which these elements interact for a strategy as below, in *Figure 2-1*. This is simply an overview of how various elements are connected to get to bet execution rather than a literal or comprehensive representation. As we will see in Part 2, there are efficiencies to be gained at many stages in the actual process, and the "building blocks" used can be combined within one program rather than several.

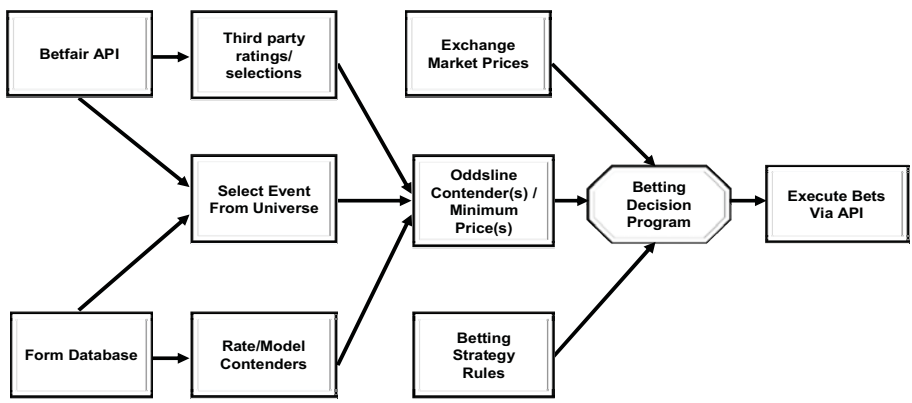


Figure 2-1: An Automated Betting Process

If we consider that each bet or series of bets comes from a single event, the starting point for the process is retrieving the relevant event details for the betting strategy from the universe of events available on a daily basis. In practice, we retrieve all applicable events and event details from Betfair at the start of the day, rather than one at a time, and generate our own events schedule, which is then accessed at scheduled times through the day.

The events inputs can come from the Betfair API, a form database (of a type that is updated automatically in order to avoid manual intervention, as we discuss in the next Chapter), or a combination of both.

Next we assess contenders within the race. Depending on the strategy, this may be done by accessing the form database and programmatically applying an existing model, or there may be third party input to retrieve (for example from a website or external ratings system).

Depending on the strategy and source of selection inputs, these are fed to the process that deals with formation of an oddsline to use as a basis for betting decisions.

The Betting Decision program takes the oddsline, retrieves market prices and applies a given set of rules or betting criteria - depending on the strategy - to determine whether or not to make a bet, on which selections, and on at which terms.

This information is fed to the betting execution program which executes bets and retrieves bet details for storing to a database.

We cover all these stages in more detail within Part 2, *Implementing An Automated Betting Framework.* Next, however, we will look at what tools we need to implement that Framework.

Chapter 3 : Tools of the Trade - Hardware, Software and Data Sources

In this chapter we will consider the IT environment for developing, testing and running completely automated betting strategies. We can break this down further into hardware and software components needed, as well as data sources.

The set up is focussed on the requirements necessary to run the type of strategies and examples in this book, where the emphasis is on complete rather than partial automation or running third party applications, putting control of the process entirely in the hands of the bettor's programs.

Using this set up, almost any modern PC can be repurposed as a betting machine, dedicated to running betting strategies which are proprietary to its owner.

In terms of software, we focus on widely used, powerful, freely available and well documented open source software tools, where many texts also exist for the beginner.

Whilst making some specific choices for the examples used in terms of Linux, Perl and MySQL, we also consider that these can be used in heterogeneous environments (i.e. within existing Linux, Windows and Mac systems). Further notes on installation and configuration are provided in Appendix 1.

In terms of the logic described for the examples in Part 2, it should also be clear that they can be readily adapted to other programming languages and operating systems in order to achieve similar objectives.

Finally, since none of the IT infrastructure is of use without appropriate data available on a timely basis, we look at the types of data that are required and typical sources for such data.

Hardware

At a basic level a computer and internet access is all that is needed, although the speed of both makes a difference to the effectiveness of any strategy. The hardware set up also needs to be such that the computer can be left unattended. Therefore, the hardware itself should be reliable enough that it can always be left on (on appropriate powersaving options, so that programs run as background processes), depending on the time that the bettor will be absent.

Ideally, there will be a back up strategy and media, as well as an Uninterrupted Power Supply to ensure the PC continues to run and back up programs and data in the event of program and/or power failure.

Alternatively, the bettor can set up their configuration within a hosted service, where the hardware is maintained by a third party.

Personal Computer:

Given the recommended development environment, most desktop PCs or laptops that are newly or recently acquired will be more than up to the task of running automated betting strategies, although a high specification is always useful. However, a more interesting question can be repurposing older, less expensive PCs as standalone betting machines without any significant loss in performance.

At a minimum, for the examples shown where we will be running Perl programs and connecting to the Betfair API, the PC should have sufficient memory, CPU capacity and disk space for a comfortable installation of Linux, Perl and MySQL – meaning the programs (all light overhead) can be executed rapidly and the data needed for automatic betting can be stored. Staying safely above the *Recommended Minimum Hardware Requirements* for one of the latest releases of a specific Linux distribution – such as *Ubuntu* (*www.ubuntu.com*) - would be a fair benchmark to use. For specifics, go to (for example) *https://help.ubuntu.com/community/Installation/SystemRequirements*. Of course this benchmark should be scaled where significant amounts of data are to be stored and accessed, or the bettor's programs become significantly more computationally intensive than the examples used.

Infrastructure for internet access:

Speed of connection to the Betfair API is paramount in execution, for capturing market prices and volumes with the minimum of delay, similarly in executing bets. Therefore the fastest, most reliable "always on" broadband option that is affordable is recommended, as is the most reliable, responsive Internet Service Provider (ISP), since routing of requests from the ISP can be an issue with regard to fast access to the exchange.

The extent to which an automated betting strategy has access to fast, reliable internet access generally has far more impact on the performance and efficiency of an automated strategy than any improvement in the specification of the hardware (over and above the recommended minimum).

Even with such a set up, there will most likely be occasional issues, e.g. the connection can be lost due to issues with the line provider, the ISP or the bettor's own infrastructure.

Strictly speaking, a dial up modem can also suffice for activity which is not particularly time critical and for basic betting strategies which just involve "getting on", perhaps at Betfair SP, but this is a sub-optimal set up which will limit possibilities.

Back up policy and media:

Whether this is an automated back up procedure to external media devices (e.g. dedicated external hard drive or other back up media) or intermittent manual back up (e.g. via a USB flash drive), we will repeat the usual caveats associated with maintenance of any useful system: programs can take a long time to develop, data can take a long time to collect, and both can take a very short time to disappear.

Back ups can generally be scheduled automatically via the operating system, (e.g. using efficient utilities such as **`rsync`** in Linux to incrementally back up only files that have changed).

Uninterrupted Power Supply (UPS):

For development and testing, a UPS may be a nice to have, but if implementing a production strategy, the bettor will want to minimize the impact of predictable risks such as power failure. An easy way to mitigate such problems is by connecting an Uninterrupted Power Supply for the PC, with a battery life that budgets can afford.

Third party hosting services

This section has focused on hardware requirements for personal use of custom automation programs, assuming a complete "Do It Yourself" approach.

Another option is to use a third party, hosted server environment for configuring set up and automating betting strategies, particularly for prediction.

This may be attractive where there are glitches in the hardware environment or internet infrastructure used by the bettor; however, maintaining a viable home environment will still be necessary for connecting to the third party server for maintenance and possibly remote back up.

Software

In terms of choosing an appropriate software environment on which to develop and then run automated betting strategies, there are a number of viable choices, without a right or wrong answer.

An idiom from horseracing is apt, being a case of “horses for courses”. In other words, the choice of software may depend on the betting strategy and the tasks that come from that or simply upon the developer’s preferences in terms of what they already know.

We should be aware of some constraints in that the operating system, choice of programming language(s) and database management system should be fit for the purpose we have in mind, including scheduling tasks via the operating system, extensive data manipulation and so on, which we will cover in the sections below.

Additionally, the operating system and any programming languages or applications used should “all play nicely together”. This means that there are no interoperability issues, and all tools taken together enable us to achieve what we want when it comes to setting up an automated betting environment, rather than considering each piece in isolation.

To run our examples we use well known and well supported tools (both commercially and by the open source community), that lend themselves very well to the principles of automation in computing. Many products fit the bill within this remit, amongst them Linux as an operating system, MySQL as a database and Perl as an interpreted programming language. These are now mature open source products used by millions throughout the world, with stable releases that can be freely installed, excellent documentation, and many texts for the novice as well as advanced user, some of which are listed in the Bibliography.

They are also increasingly used as an *ensemble*, (as in the *LAMP* environment) and most Linux distributions (or “distros”) will include a compatible release of Perl and MySQL as part of the distribution, meaning that the effort required to set up the whole environment is reduced. Despite the mention of Linux as the operating system, many options now exist to run Linux on PCs with an existing Windows installation, and Perl and MySQL have stable Windows releases, so if required, the examples can be adapted for Windows with some tweaking and further work, as we discuss in Appendix I.

Operating Systems

Since the Operating System (OS) manages access to the computer’s resources, controlling all processes and enabling them to work with each other, including automatic scheduling, the choice of operating system is as critical as the programs used. Prime concerns for the automatic bettor are reliability, security, ability to troubleshoot easily, flexibility and, most of all, support for automation.

Typically, our existing OS in a home computing environment will be Linux, Apple Macintosh or Windows, or a combination in the case of dual boot systems. On a Mac

nowadays, with any version of OS X, the operating system is also UNIX based, accessible via the Terminal application (and now also uses an Intel chip).

Linux (within which definition we include the Mac OS X family, being based on Unix and equally able to run our example programs) is therefore the chosen operating system for the examples in the book for all the above reasons, at a minimum to facilitate excellent support for scheduling, running multiple processes and reliability.

For the purposes of converting our machine to a tailored automatic betting environment we are usually working from the command line. The Linux environment gives us such a shell (or command line) environment with powerful UNIX commands at our disposal. We will also adopt the UNIX philosophy of creating small functional programs or utilities that can be used and tested on a standalone basis to fulfil elements of our automated framework for the betting process, and then linked together to create automated applications. Adopting the idea of creating small, useful programs or scripts also makes for ease of prototyping and maintenance, since it can be easier to write, diagnose and fix a small program, or replace it, than to do the same for a big program.

From the point of view of automation, we have the flexibility of the powerful *cron* utility and the Linux *at* command, as well as the possibility of creating daemon processes. Linux gives us this as part and parcel of the operating system, as well as playing nicely with other applications and programming languages, such as Perl, which was created initially on a UNIX platform.

Whilst the Windows OS is ubiquitous, it is not our first choice OS for the bettor setting up a betting machine at home, since in the ideal case we want to leave systems to run reliably unattended for long periods of time.

In this sense, Windows and Linux come from different design perspectives, both making different assumptions about the way home users wish to interact with computers, for different use cases. The emphasis in Windows is creating out of the box applications with emphasis on interaction through the GUI, whereas coming from UNIX, Linux is about reliability, control through the command line and enabling multi-user production systems with security built in. For the automatic bettor who wants to focus on optimising their betting strategy, automation via the OS should “just work”, rather than spending time ensuring the appropriate Windows virus software is up to date, for example. Similarly, if there are any issues they need to be able to “get under the hood” and address them quickly in order to avoid downtime in betting strategies.

Having pointed out some of the potential drawbacks with using a Windows OS in this way, we should also note that there are many other options to install Linux to sit alongside Windows as a dual boot system, or even as a virtual machine within Windows, so it is not necessary to change computers if going down this route. The examples can also be adapted to run natively on Windows if required.

For those who already use Linux, they will have their own favoured distribution, under which all the coding examples should work, provided the correct Perl modules and MySQL are installed (as detailed in Appendix 1). For those who do not currently use Linux, the distribution recommended is Ubuntu (installation notes also in Appendix 1).

The reasons for recommending Ubuntu are the ease of install and support for the OS itself, automatic partitioning for dual boot with Windows (if required), and the availability of Perl, relevant Perl modules and MySQL versions which can be installed directly from the Package Management System, as if “out of the box”. Also, Ubuntu is now a supported OS that is supported pre-installed with new computers by some major PC manufacturers.

Finally, whether currently a Windows, Mac or Linux user, it is well worth considering setting up a separate betting machine or server in the long term purely for running automated strategies, since this can be maintained independent of a home PC that may be in constant use for other applications.

Programming Languages

As with Linux, where any existing user of Linux will have their own favoured distribution, so any existing developer will often have their own favoured programming language(s).

Perl is used for the examples in this book both as a common language to use for explaining the principles of automation and interaction with the API, as well as being a useful starting point for non-developers who want to explore the process, where plenty of literature is available to pursue further.

There are many other well-respected scripting languages with similar attributes (and of course advantages and disadvantages) such as Python and Ruby, leaving aside compiled languages for a moment, which can also be used in this context. For those who wish to pursue other routes, the intention is that the examples can serve as pseudo code to illustrate some concepts and methods of implementation.

However, the attributes of Perl – which stands for “Practical Extraction and Reporting Language” – serve particularly well in automating the betting process, since very often we are simply trying to extract relevant information in one part of the process, clean it, repurpose it, and move it to another part of the process, ensuring that the data can be compared on a like for like basis at each stage. Perl provides high and low level tools to help the user accomplish such tasks quickly.

For example, at a basic level we are often faced with the problem of matching relevant information – such as a horse name on Betfair that is missing an apostrophe or a capital letter and has to be matched to a horse name on a form database which has it included. Or, we may need to process a stream of unstructured data from a website and bring some order to it. Or, in the case of the Betfair API, we need to work at a low level to retrieve data in XML and parse the returned data. Perl has few equals for such text processing and data manipulation tasks.

However, the task of betting automation is not data manipulation alone. Another important area for us in terms of scheduling, communication and process flow, and again Perl is a great “glue” as an all purpose language. Additionally, any programming language must enable the bettor to easily write rules and algorithms for decision making - for example determining what to bet on and how much to bet.

Although this may seem the crux of any program, it is often a tiny fraction in terms of percentage of code and time spent. In any case, programming complex algorithms, where needed, also fits well within Perl's compass, tying in with the Perl philosophy of "Making Easy Things Easy and Hard Things Possible".

In the case of harder to code functions and algorithms, Perl benefits from being one of the more mature and popular languages available so that there are many contributed extensions or modules which can be added to a standard Perl installation that already accomplish such tasks, mostly available at the Comprehensive Perl Archive Network *http://www.cpan.org*. A number of such modules will be used for the example code provided, and those that are not installed as standard are listed in Appendix 1.

Last but not least, despite being an interpreted language, the speed of execution in our Perl scripts is excellent for the purposes of betting automation. This too, is an important element to consider in choosing any other programming language for this use case.

Database Management Systems

There is a need to capture, store and reuse data throughout the automated betting process. This may be recording oddsline selections, events data and corresponding bet details (from Betfair) for further analysis, or to make data persistent for use between different programs. Persistent data elements might include events, prices, Betfair runner identification numbers and so on. Whilst some of these elements can be passed from program to program as flat files or simple "key-value" pairs, it is clear we should be storing some variables that are required on a long term basis – such as detailed records of all bets placed - in a database for ease of management and reporting.

Moreover, where the bettor is running programs to analyse form and rank contenders ready for submission to the rest of the betting process, there will of course be a necessity for an up to date form database from the beginning of the process. We will come back to the case of data sources for the form database under the "Data Sources" section, although the comments here on the type of database management system to be used are also applicable

It is likely that as automated betting strategies grow in complexity, and the number of transactions grows in volume, working with a database will play an increasing role in any strategy. Further, if the bettor wants to enhance strategies by re-using some or all of the data which is collected on a daily basis, rather than simply discarding it, the size of the database will grow rapidly. For example, for research into price movements, a database will be essential to capture and store exchange price information for later retrieval and analysis.

However, perhaps the most fundamental requirement for using any database management system within the automation framework is that it should be possible to automate all database operations, ideally via an interface to a programming language. This means that inserting data and querying it can be performed automatically. Once data is automatically extracted it can be analysed in other programs or passed to other parts of the betting process, depending on the purpose of retrieving it.

This is not always the case with proprietary database formats, where data manipulation and automated queries via non-proprietary languages may be unsupported and unreliable, without time consuming and specialist programming effort. Any lack of interoperability wherever such a database is used or lack of common standards can lead to a serious breakdown in the automation process. To keep a proprietary database within the set up which does not "play nicely with others" could require a stage of manual intervention at every turn, and render the objective of total automation in the bettor's absence null and void.

Fortunately there are a number of highly robust databases now available on an open source basis, two of the most popular being MySQL *www.mysql.com* and Postgres *www.postgresql.org*. These databases are more than sufficient for most applications which the bettor would wish to implement. Moreover, many programming languages include supported interfaces to them, including Perl's database integration module, the DBI. This means programmatic control of database operations is achievable, and can therefore be scheduled or automated.

The references made in this book focus on MySQL which is a symptom of its popularity, documentation, ease of use and interoperability with Perl. However, the examples can easily be adapted for other databases.

Data sources

The need for relevant data persists through all stages of the betting process: historic form for assessing contenders, daily racecard data for identifying suitable events, market data to assess prices, account data for money management – the list goes on.

The choice of data sources will typically be determined by the bettor's particular strategy (i.e. by the data needed to execute the strategy) and where more than one appropriate source is available, the cost and/ or format in which the data is available. For automation, we rely in all cases upon data being available in some electronic form. Typical sources that might be of use include:

- Betting Exchange Data
- Commercial Form Databases
- Online Form, Results and Racecards
- Market data from Fixed odds, Spread firm and Odds comparison websites
- Miscellaneous websites – anything from tipster selections to sectional timing statistics and meteorological reports to predict going

This is not intended to be an exhaustive list, although the vast majority of data that can be of use falls into one of these categories.

For many stages of the betting process, the choice of data source and the format is clear cut - particularly for retrieval of Betfair data elements via the Betfair API. However, navigating for other sources, accessing and using the rest of the cornucopia of electronic data available for automatic betting programs is another question again. Within the text we provide some specific references where appropriate, but first we consider the question of data formats and programmatic access generically.

Data Formats

Technically, there are challenges to overcome with many of the formats used for the data sources listed, which typically fall into one of the following categories:

- Website technologies (HTML, XML, Javascript etc)
- Proprietary applications, including databases and spreadsheets
- APIs
- XML feeds

Screen scraping – retrieving data from the web

Taking for a moment the case of the web, any data required from the body of website content requires us to write a webscraping or screen scraping program, which involves custom work to retrieve data automatically, from parsing HTML to managing cookies and handling forms. Each program must be tailormade to the website in question.

All these challenges can be overcome, since most modern programming languages have high level modules to somewhat simplify the process, including Perl.

However, automatically parsing text, cleaning and polishing it to retrieve the exact data elements required will always be a job of work.

In the case of using Perl, our recommended programming language, Perl's LWP (Library for the world wide web in Perl) is a collection of modules for accomplishing the task of creating a program in Perl to extract and parse any data that is normally accessible via a browser. Perl's modules are organised to enable programs to download and extract information from the Web on an automated basis, including automated navigation, submitting forms, and providing authentication information (i.e. for sites requiring login and password details). In Perl's case, such modules build on Perl's unparalleled reputation for text processing and as an all purpose "IT glue".

On the other hand, whatever tools are at the developer's disposal, websites provide a very fragile interface for data extraction, primarily because they are designed for user interaction rather than screen scraping applications. Thus, the organisation, display and information available on any given website is subject to change without notice. This means that any such programs are by definition unreliable and in potential need of constant maintenance.

Moreover, whilst almost anything we can do on the web interactively can also be automated, that is not the same as saying that any data found on the web can be freely used, or that access to all sites is permitted (even if it is possible) via automated programs.

Terms of user interaction are different from terms of programmatic interaction and making "electronic copies". At the end of the day, websites dictate their own terms of use and the automatic bettor must satisfy themselves that such terms are covered.

Commercial Form Databases

For extensive research on results, predictive modelling, systems creation and programmatic assessment of contenders, there is no real substitute for using a reliable source of form data, which will generally mean the bettor maintaining their own form database. In fact, unless strategies rely only on third party sources, it is to be recommended. Generally, website retrieval of form data is not reliable or legitimate for such uses, and besides, there are a large number of useful providers available

However, commercial form databases can pose similar problems to website access for the automatic bettor with regard to data formats and the purpose of use. Without attempting to review the many commercial databases available, let's assume for now that all these sources and others not mentioned contain the data attributes we need to run betting systems or build predictive models, (although clearly that is a case for the buyer to validate).

There is no issue with legitimate access, but there can be issues to automatically extract data stored within proprietary database formats or applications for use in programs without extensive custom work. Even where such facilities exist to export data from some databases, automating the process is another issue.

Another challenge is to ensure that form is automatically kept up to date, and racecards can be evaluated in time for daily betting strategies. The delivery mechanism is therefore a key concern for the automatic bettor, since this should be reliable but also compatible with an automated process which requires no manual intervention.

However, there is light at the end of the tunnel, since at the other end of the spectrum, various formats (which are used on an increasing basis) are specifically designed to enable automated access – or designed to be "machine readable".

The brave new world of data access is signalled by the latter formats on our list, namely APIs (application programming interfaces), XML (extensible markup language) feeds, and, in terms of local installation, many open source applications which can be updated and provide for programmatic access.

The accompanying website to the book provides its own contribution to the list in the form of an automated results and racecards database, *SmartForm* for use with MySQL - *www.betwise.co.uk/smartform.html*.

It is not just the technology behind the use of open standards which is useful, but the thought process behind the technology. When these formats are used it is often with programmatic access at the front of the list. This makes life ideal for automating betting.

The Betfair API

Whether we generate systems for ourselves or use other inputs for the purpose of selections or trading, the majority of the process is still left to do and requires interaction

with the exchange, extracting market information, making a decision based upon that information, and then placing a bet. The same is true for record keeping after placing bets.

Even runner and all basic race information can effectively be gleaned from here, without using other sites. In fact, since the rules of each market depend on it, Betfair is often quickest off the mark with regard to listing non-runners, so is a good port of call for that from an up to date racecard perspective. Therefore, much of our automated betting process relies on interaction with the Betfair API.

It is worth mentioning at this point that APIs exist for other exchanges, notably Betdaq *www.betdaq.co.uk*, but we focus on the specifics of Betfair since it is by far the most liquid of the exchanges, and the Betfair API is particularly well supported. Nonetheless, the principles of implementing automated betting strategies are applicable to exchanges in general.

Now, it is technically possible to capture this information by screen scraping, as we have described earlier, and if that were the only method available of capturing and using the data, then we would be including all the caveats associated with that process. Fortunately, the Betfair API provides us with a reliable well documented interface which is not subject to changes in the website, outages, or changes in access policy.

In short, the API is designed to enable reliable machine to machine access of Betfair data and functions, which is what we are interested in as automatic bettors. The Exchange API is independent of the Betfair web interface; therefore the bettor can build custom programs to suit their betting strategy.

In the next part of the book we will cover the creation of reuseable libraries and scripts with which to implement the different stages of the betting process, starting with a more thorough overview of how we can use the Betfair API programmatically.

PART 2: IMPLEMENTING A FRAMEWORK FOR BETTING AUTOMATION

Chapter 4: Using Betfair API services

In this chapter, we will consider what Betfair API services are, the different services available, and ways of calling those services using Perl. Having considered how Betfair API services work, we will discuss creating Perl functions to make those services available to programs, using an example library of functions. The example library is printed in Appendix 2 and is also available for download at *www.betwise.co.uk*.

Since the Perl functions are stored in a library which can be called at the start of other programs, these functions can be maintained separately and reused by any Perl program which is implementing a betting strategy thereafter. The library therefore provides us with one of the essential building blocks that we will use again and again in implementing different parts of the automated betting framework, from selecting available events to bet in to retrieving details of bets made. So this chapter will deal with implementing the library that deals with the API generally, and in subsequent chapters we will look at example programs to implement specific parts of the betting process, which - in the case of programs accessing information from Betfair or performing transactions (such as executing bets on the Betfair exchange) - will also use this library.

First, a word on semantics – the terms call, service and function essentially refer to doing the same thing but from different perspectives. The Betfair API provides a number of services, which can be called by client programs. Functions, or subroutines in the programming language, may implement those calls in order to get results (i.e. data) back from the Betfair servers and process that data, so that it can be used generically within any given program. This is what our library of Perl functions, or subroutines will do for each API call.

How the Betfair API works

For those familiar with the Betfair website interface but new to the Betfair API, or new to using any API, this section provides a general overview of how the Betfair API works.

At a basic level, the website and the API are simply alternate means of accessing the Betfair Exchange. The same exchange that many users are familiar with accessing interactively on the website can be accessed programmatically via the SOAP (Simple Object Access Protocol) API, as represented in *Figure 4-1*:

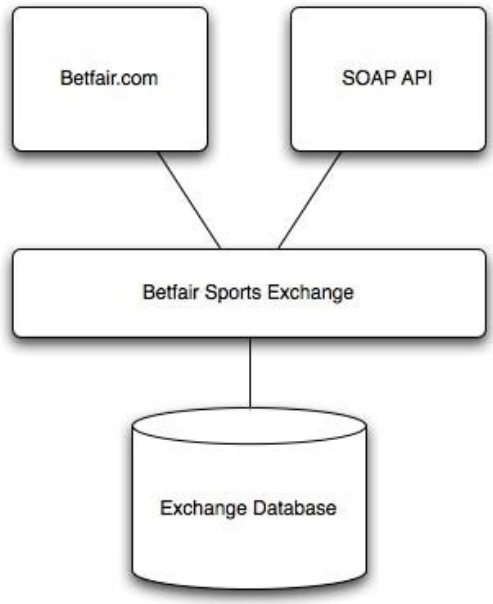


Figure 4-1: Communicating with the Betfair Exchange

Whatever mechanism is used to access the exchange - website or API - the starting point is a client program. In the case of using the website, the client program is simply our web browser of choice. In the case of the API, the client program is up to the individual developer, or the company providing an application which has been developed for the user.

For a client program to access the exchange by using the API, the client program must send the API an XML request using the SOAP protocol to transmit that request. XML (extensible mark up language) and the SOAP protocol are simply the format required to transmit messages reliably (we cover the calls available and the use of XML in subsequent examples in this Chapter). The Betfair servers process and validate the XML request and retrieve the appropriate data from the Betfair Exchange. The Betfair Exchange returns the information requested which the API packages into a properly formatted XML response and sends to the client – it is up to the client program to decipher this response. Our functions will make the validated requests to the exchange and decipher the response.

We can therefore see the website and the API as two different “front ends” to the Betfair Exchange. The processing of any bets, including matching, ordering, etc all happen within the Betfair Exchange, not in the client program or in the ether between the exchange and client program.

Speed of execution for any client program using the API is greatly accelerated by eliminating manual point and click processing, as well as menus that the programmer may deem superfluous. In the case of complete automation, there is no need for production programs to interact at all with the user, so the speed at which betting strategies can be executed is down purely to connectivity.

In terms of speed of connection to the exchange generally, which can be critical for many strategies, in particular trading strategies, there are many exogenous factors to consider, some of which are: caching differences, delays in network protocols, ISP routes to Betfair, browser speeds, SOAP proxy libraries. With the functions shown as examples, any overhead in processing is confined to milliseconds. Usually, if there is an issue with speed, this is down to the user's network and ISP routes to Betfair - but here it should also be possible to reduce delays to milliseconds.

Description of Betfair API services

The Betfair API has different service calls available, and levels of availability for those services, depending on the Betfair account holder's subscription to the API. All Betfair users have access to the Free Access API (provided they have an "active" account, defined below). Access to the Free Access API can be enabled in client programs by specifying a product code of "82" to the **Login** service, together with the Betfair account username and password. Most of the available services within the API are accessible at the Free Access API subscription level, as can be seen from the table listing which calls are available at each subscription level in Appendix 3, *Betfair API Services*.

The service level for certain calls varies in terms of the number of times per minute that a call to a Free Access API service can be made. When a call is restricted this is referred to as "throttling". Throttling limits for each of the Free Access API calls are also shown in Appendix 3. However, no automated betting strategy or example shown in the book relies on any service which is not available via the Free Access API, or which must be called with greater frequency than the throttling limits available, although we do refer to cases when other services and call frequencies are useful.

Essentially many useful strategies can be executed 'as is' on the Free Access API, and many others tested in principal. Users may at some point decide to change their level of access, either directly with Betfair (for general development access) or with a Software Vendor supporting specific applications, or programs to be developed. Generally such considerations will be down to running more intensive betting strategies (e.g. applications which must access the exchange multiple times per second), how much the user wants to do themselves, and the availability of support.

The Free Access API requires that the user's account is "active", i.e. that they have deposited some money, placed a bet that's been matched and settled, etc within the last three months. If they haven't the API will return "PRODUCT_REQUIRES_FUNDED_ACCOUNT" and the login fails.

In order to implement a successful automated strategy, it should be noted that it is not necessary to use all services, or even the majority – a few key services crop up again and again, whilst others may be hardly, if ever, relevant. For example, logging on, obtaining market information (event and runner details), then market prices, and bet placement can form the foundation of many successful strategies, albeit used in different combinations and on a repeated basis. For such strategies, there are only 4 or 5 different services needed. There is also more than one way to skin a cat by using different services for the same purpose. For example, we show how to use 2 services in concert when retrieving the list of all horseracing events in the UK and Ireland in the next Chapter. The **GetAllMarkets** call would be an alternative way of doing this.

In the above sense, we can look at the services like an analogy to language. The availability of words that can be used in any language far exceeds common daily usage, or even common knowledge – and yet we can express complicated concepts for which other words exist by using a mixture of those commonly used. We will see this more closely when we look at example strategies in Part 3.

In the meantime, a list of the most commonly used services available from the API is provided with a corresponding description in the table below. This provides a useful reference by which to think about calling services in different combinations in order to implement different strategies.

Table 4-1: Description of Betfair API Services

Name	Description
Login	The API **Login** service enables customers to log in to the Exchange API service and initiates a secure session for the user. Users can have multiple sessions alive at any point in time.
Logout	The API **Logout** service allows you to explicitly end your session.
KeepAlive	The **keep alive** service can be used to stop a session timing out. Normally a session is expired if it has been idle for a period of time. Issuing periodic KeepAlive requests will stop this occurring.
GetActiveEventTypes	The Exchange API **Get Active Event** Types service allows the customer to retrieve all Sports (Games, Event Types) which have at least one active/suspended market associated with it currently. This service would for example always return the event types Soccer and Horse Racing but would not include Olympics 2012.
GetAllEventTypes	The Exchange API **Get All Event Types** service allows the customer to retrieve all Sports (Games, Event Types) regardless of whether they have an active market associated with them. This service would for example always return the event types Soccer and Horse Racing and would also include Swimming, Olympics 2004 or EURO 2004 even when the events have finished or have no current events. This is to allow API programmers to be aware of the full range of potential sports that may be available in future.
GetEvents	**GetEvents** is used to navigate through the menu structure. It allows you to input a Sport or Event and retrieve all Events or Markets which have the input event id as a parent.
GetMarket	The API **Get Market** service allows the customer to input a Market ID and retrieve all static market data for the market.

Table 4-1: Description of Betfair API Services (continued)

GetAllMarkets	The API **GetAllMarkets** service allows you to retrieve information about all of the markets that are currently active or suspended on the given exchange. You can use this service to quickly analyse the available markets on the exchange, or use the response to build a local copy of the Betfair.com navigation menu. You can limit the response to a particular time period, country where the event is taking place, and event type. Otherwise, the service returns all active and suspended markets.
GetMarketPrices	The API **Get Market Prices** service is used to retrieve dynamic market data for a given Market ID.
GetMarketPricesCompressed	The Exchange API **Get Market Prices Compressed** service is used to retrieve dynamic market data for a given Market ID. This service returns the same information as the Get Market Prices service but returns it in a ~ (tilde) delimited String.
GetAccountFunds	The Exchange API **Get Account Funds** service allows the customer to retrieve financial information about their account.
GetDetailAvailableMktDepth	The **Get Detail Available Market Depth** service returns the current odds and available bet/lay amounts on a runner in an event.
GetMarketTradedVolumeCompressed	The API **GetMarketTradedVolumeCompressed** service allows you to obtain the current price (odds) and matched amounts at each price on all of the runners in a particular market.
GetCompleteMarketPricesCompressed	The API **GetCompleteMarketPricesCompressed** service allows you to retrieve all back and lay stakes for each price on the exchange for a given Market ID in a compressed format. The information returned is similar to the **GetDetailAvailableMarketDepth**, except it returns the data for an entire market, rather than just one selection.
GetSubscriptionInfo	The **Get Subscription Info** service returns information on your API subscription.
GetAccountStatement	The **Get Account Statement** service allows the user to obtain information on transactions that have occurred on an account.
GetCurrentBets	The **Get Current Bets** service allows the user to retrieve information about bets that have been placed. Information can either be retrieved from a single market or across all markets. This request supports paging through the result set through the use of the **Starting Record** and **Record Count** parameters.
GetBet	The **Get Bet** service is used to retrieve information a specific bet that has been placed. Each request will retrieve all components of the desired bet.
GetMUBets	The **GetMUBets** service allows you to retrieve information about all your matched and unmatched bets on a particular exchange server.
GetMarketProfitAndLoss	The **Get Market Profit and Loss** service allows you to retrieve Profit and Loss information for the user account in a given market. The limitations for the service in the initial release are: - Profit and loss for single and multi-winner odds markets is implemented however it won't calculate **worstCaseIfWin** nor **futureIfWin**. - The calculation for Asian Handicap markets will include **worstCaseIfWin** but not **futureIfWin**.
GetMarketTradedVolume	The **Get Market Traded Volume** service is used to obtain all the current odds and matched amounts on the runners in an event.
GetAllCurrencies	The Exchange API **Get All Currencies** service allows the customer to retrieve all the available currencies and their exchange rates.
WithdrawToPaymentCard	The API **WithdrawToPaymentCard** service allows you to withdraw funds from your UK wallet using a previously registered payment card.

Table 4-1: Description of Betfair API Services (continued)

DepositFromPaymentCard	The API **DepositFromPaymentCard** service allows you to deposit funds into your UK wallet from a previously registered payment card. You cannot deposit funds directly into your Australian wallet.
TransferFunds	The **TransferFunds** service is for transferring funds between your UK and Australian account wallets.
ConvertCurrency	The Exchange API **Convert Currency** service allows the customer to convert a currency
WithdrawToPaymentCard	The API **WithdrawToPaymentCard** service allows you to withdraw funds from your UK wallet using a previously registered payment card.
DepositFromPaymentCard	The API **DepositFromPaymentCard** service allows you to deposit funds into your UK wallet from a previously registered payment card. You cannot deposit funds directly into your Australian wallet.
PlaceBets	The Exchange API **Place Bets** service allows multiple bets to be placed on a single Market. There is an instance of PlaceBetsResp returned in the output for each instance of **PlaceBets** in the input.
UpdateBets	The Exchange API **Update Bets** service allows editing of multiple bets on a single Market. There is an instance of UpdateBetsResp returned in the output for each instance of **UpdateBets** in the input.
CancelBets	The Exchange API **Cancel Bets** service allows you to cancel multiple bets on a single Market. There is an instance of **CancelBetsResp** returned in the output for each instance of **CancelBets** in the input.
CancelBetsByMarket	The Exchange API **CancelBetsByMarket** service allows you to cancel all unmatched bets(or unmatched portions of bets) placed on one or more Markets. You might use this service to quickly close out a position on a market.

A library of Perl functions to call the services

So we know how the API works in principal, we have a description of the services available, but we have still to make the connection between the Betfair exchange and the client program so that automation can happen. For this we need to use a programming language, which applies whether we are using a point and click, semi-automated application, or one that runs invisibly to the user.

The mechanism by which the Betfair API services are provided and accessed so that the data extracted can be used on a client machine is broadly defined as web services. We are retrieving information from a specific internet server using protocols designed for the secure and reliable transportation of data, in this case the SOAP standard, which is a wrapping mechanism for passing data defined in XML format.

There is a processing overhead on each side to manage such tasks. We must make any request by complying with the service protocols used, specifying the correct parameters and XML format which the service will respond to, as generally defined in the WSDL (Web Services Definition Language) for whatever service we are interested in accessing. Additionally, we must process and interpret the results of the call, which are returned in XML format, within our function. This is taken care of in the example library.

Perl, as with many languages, has high level user contributed functions and modules to make the task of using web services easier, as well as being able to handle the process from a low level (without specialist modules taking care of everything). For some other programming languages and platforms (such as .NET for Windows), examples of calls are also shown on the BDP (Betfair Developers Program) website.

For our purposes, we will build functions or subroutines in Perl to access each of the important services in the Betfair API. As with anything in Perl, "There Is More Than One Way To Do It", which of course includes both more messy and more elegant ways. In terms of creating functions to call API services, not only do we have a number of choices over how to write each function, we also have many choices when it comes to implementing the functions and methods of accessing them.

In this case, we will create the functions or subroutines and store the functions in a common library, or module, enabling that library to be called by other programs in the future. This is far more efficient than writing individual subroutines within individual programs – since we will use the service calls again and again and will want to maintain them in one spot. No doubt the subroutines can be improved, having evolved over time and from various Perl discussions on the Developers Forum as the API versions were revised. However, the acid test is that they are fit for purpose and can be maintained, since we are mainly concerned with implementing something that works and which is transparent. The great thing about Perl is it that the "P" stands for Practical. For the individual, making something work in the least time possible is generally the first priority.

The next subsection provides further background about how the service call functions are constructed, whilst the full script for the library of functions called in the examples is shown in Appendix 2.

Building service call functions

To understand the mechanics of how the functions work, let's take a quick look at the essential elements to be considered when building such functions within any language.

First, we should refer to the comprehensive technical documentation for the Betfair API provided at the Betfair Developers' Forum, in particular the main source of technical reference for using API services, which is the *Betfair Sports Exchange API 6 Reference Guide* (or subsequent versions as they arise). This is freely available for download (along with other useful materials) at *http://bdpsupport.Betfair.com* (then click on the *Downloads* link under the *Support Centre* tab).

This is the main technical reference for creating and troubleshooting service calls, although of course the documentation relates to the API only rather than any specific programming language. Some general background is provided, but in the main all of the documentation refers specifically to the parameters of each API call using the following structure:

Service name: Specifies the name of the service to be called and whether the service is available through the Global or Exchange URL endpoints (i.e. server addresses).

Input: Table showing which parameters have to be specified in order to call this service, and a full description of those parameters.

Output: Table showing which parameters are returned by a successful call to the relevant service, and a full description of those parameters.

ErrorCode: Table showing error codes returned in the event of an unsuccessful call, and a full description of those codes.

The official documentation provides generic information about the data structure passed to and from the service; implementing that within any given program and deciding how to retrieve or present the information returned is all left for the user of the API. This is precisely what our library of functions should take care of.

In the next sections we'll take a look at a couple of example functions from the Perl library attached with the book (without repeating the process for all) to get a flavour of how they work and are put together. The library of which the functions are part is just one way to do it, so can be readily changed, adapted or improved to suit any user's particular preferences.

A quick note on naming conventions used: each example Perl function (from the library of functions `BetfairAPI6Examples.pm` listed in Appendix 2) shares the name of the Betfair API service it is accessing, but the Perl name is in lower case (whereas all words in Betfair services are capitalised) and separated by underscores for each word (whereas there are no spaces by convention in the Betfair API Service names).

For example, the Betfair API Service **GetMarketPricesCompressed**, finds the parallel Perl function which calls it in `get_market_prices_compressed`. Sometimes the Perl function will also appear in bold for emphasis, but always in lower case and in a monospace font.

First we will look at the most fundamental example function, **`login`**, for logging into the exchange, (since that is a prerequisite for making any other service calls) followed by an example covering one of the most frequently used functions to retrieve market prices, **`get_market_prices_compressed`**. Since the structure of each function is similar, many of the comments regarding the code for the **`login`** function and the **`get_market_prices_compressed`** function apply generally to all the functions in the library. Finally, without further ado, we'll list and summarise the inputs (or arguments) required for the other example functions in the Perl library, together with the outputs from those functions – in other words, we will provide a summary for setting the other functions to work.

Review of the `login` example function

The majority of work in all the API function calls is ensuring that our code is passing valid XML requests and using the specific protocols and mark up language (XML within a SOAP Envelope) that is required for each service we want to call, together with the correct parameters for that service. Once the call is successful, we need to process the

XML that is returned from that service, since it contains the information we want, and put that in a form where we can readily access it. Combining these elements will subsequently enable us to call the subroutine in the context of any relevant program, as we discuss for some of our later example betting strategies.

Thus, when reviewing the functions there is a spaghetti soup of required data and mark up tags confronting the reader. It's useful to bear in mind our objectives when faced with this. With regard to this section, and the `login` function, our specific objective in writing and using the service call is to obtain a so-called **sessionToken** that we can use to make any subsequent service call, be that getting market prices or placing a bet. The session token is a string containing encrypted information, by which to authenticate any future service request. The session token is all we're after - but we can't use any other API function without it.

A couple of further points about the **sessionToken** that are worth bearing in mind when running and modifying programs:

- Each session token is only guaranteed good for a single API request. Therefore, each subsequent service that is called after login contains a session token (which can be retrieved from the response header) that must be used in the next service request.
- Usually the **sessionToken** actually remains the same from one service to the next, but if we are calling more than one service within one program it is good practice to check after each service whether the session token is the same or has changed and use the relevant one. (N.B. In the examples shown we often use separate programs for separate calls, passing the output to files which can then be picked up by subsequent programs. It's easier to demonstrate the principles involved and the output at each stage by doing this, including error checking. For many strategies, the same results can be achieved as if using multiple calls within one program).

Before stepping through the functions, a brief note on the first group of statements within the library `BetfairAPI6Examples.pm`. These simply define the library itself, as in

```
package BetfairAPI6Examples;
```

followed by the Perl modules which are required in order for all functions to operate, as denoted by the group of `use` commands, each of which is followed by a module name. Next, we ensure that all the functions and variables can be exported to any program which requires naming each of the functions which are to be made available within a list to export, using `Exporter`. In subsequent programs, the library can be used by defining the path to the library (here our example path is `/home/aeb/lib/`, along with the command to use the relevant package of functions within the library, as in:

```
use BetfairAPI6Examples;
```

Adding further subroutines to the library is a matter of defining the subroutine and adding the function name to the list of those to be exported.

So, let's start by examining the function call **`login`** for the **Login** service, step by step, that is found in the library, in order to review how it works in principal (the full function is

printed in Appendix 2, so here we do not reprint it again but walkthrough the key elements line by line).

```
sub login
    {
  my ($username,$password,$productid)=@_;
```

To start with we are defining the name of the subroutine (**`sub`**) or function (which is called **`login`**), and then in the next line specifying the arguments that the function will take when we use it (and which must therefore be supplied when using the function in any particular program. In this case, three values are expected to be defined in any program where we use this function, namely the Betfair **username**, **password** and the **productId** with which we are accessing the API (in the case of the Free Access API that is "82").

Next we will prepare to get the **sessionToken** (the objective of this function) that will enable us to continue making calls to other services when it is put to use.

First we create a variable called **`$xml`** that will contain the valid XML structure in order to call this service successfully, to be passed within a SOAP (Simple Object Access Protocol) Envelope. The XML will reuse the arguments that we have already specified as variables within the program for username etc.

```
my $xml=
'<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <SOAP-ENV:Body>
              <m:login
xmlns:m="http://www.Betfair.com/publicapi/v3/BFGlobalService/">
                     <m:request>
                            <password>'.$password.'</password>
                            <productId>'.$productid.'</productId>
                            <username>'.$username.'</username>
                            <vendorSoftwareId>0</vendorSoftwareId>
                            <locationId>0</locationId>
                            <ipAddress>0</ipAddress>
                     </m:request>
              </m:login>
       </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
```

Next we prepare the client for sending the XML off to the relevant Betfair server, the same client which will handle the response from it. In practice, this means creating a user agent (as in the variable `$userAgent`) which can be set to work remotely and automatically on our behalf, using the Perl class for automatically doing so. This class is a subset of the LWP library (or Library for Programming the World Wide Web, being one of the modules that extend Perl, making our job much easier than it would otherwise be). The `UserAgent` class will also be our friend when it comes to interacting with other websites to retrieving data programmatically.

So, the next line creates an `LWP::UserAgent` object for us, for which we can later set various parameters in order to make the request and then process the response.

```
my $userAgent = LWP::UserAgent->new();
```

In the same way as we saved an XML message for later use within the variable `$xml` to make for easier manipulation of the code, so we now create a variable that we will use to request the **Login** service, including the address to "post" the request to. This time the variable is an object called `$request` which inherits all the methods associated with the module `HTTP::Request`. The request will go to the Betfair Global Exchange Server that provides this service at *https://api.Betfair.com/global/v3/BFGlobalService*

```
my $request = HTTP::Request->new(POST =>
'https://api.Betfair.com/global/v3/BFGlobalService');
$request->header(SOAPAction => '"
https://api.Betfair.com/global/v3/BFGlobalService
"');
```

In passing we should note that so-called "global services" (for generic functions such as logging in) are denoted by **v3** within the URL in API version 6, and exchange specific services (e.g. accessing specific markets) by **v5** within the URL. We cover the endpoint URLs and accessing exchange specific services for the Australian exchange in more detail later in this chapter.

Next we set up the methods for the `$request` object, passing in our user details, as previously saved in the variable `$xml`, so that they can be supplied when the request is posted, using the content method to our `$request` object.

```
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");
```

Next the request is made to the relevant Betfair Global Service server, using our `$request` object which is itself now a method of our user agent. The user agent dispatches the call for the service and returns a `HTTP::Response` object, saved to the variable `$resp`.

```
my $resp = $userAgent->request($request);
```

At this point we are now done with the connection to Betfair as far as this call is concerned. In other words, we have all the information we need in computer memory - stored within the `$resp` variable, rather than residing on a Betfair server (including our sessionToken, provided that the call was successful). If we were to add in the command `print Dumper ($resp)` the printed output would show us the full result of our request, written to standard output.

As a general note for all the functions calling the API, using the statement `print Dumper ($response variable)` is an indispensible utility for troubleshooting response values and parsing values from API calls. It is enabled by the `Data::Dumper` module that is specified as one of the prerequisite modules to the library. The `print Dumper` statement will show the structure of the data returned by any successful call, and therefore how we should handle that data within the Perl functions, so that it can be used by our programs.

Various `print Dumper` statements are therefore made within the code and commented out (i.e. prefixed with a hash **#** symbol so they are ignored by the compiler at run time) in the final library. Of course, these can be switched on again in order to review the full data structure returned by each call at each stage, or to incorporate data within functions that is not currently included (not all data elements from the call are extracted in the examples, just the most frequently used, others can be incorporated by consulting the API documentation).

With regard to the `login` function, if we were to use `Dumper` to print the `$resp` variable to standard output we would see that the response object is represented as a complex data structure, with a set of key value pairs, separated by the **=>** operator, with keys such as '_protocol', '_header' and '_content' representing values from our response object. At the moment, the response object contains everything returned from our response to the call, but we don't need everything. Specifically we need the data within the '_content' part of the response. So the next part of the code uses the method available for extracting the content of the call from the response object and saves it to its own variable, `$content`.

```
my $content=$resp->content;
```

Needless to say we would like to represent the variable as a Perl data structure so we can access it using straightforward Perl code - as opposed to the current tag soup of the SOAP envelope. Note, we say data since, in addition to the session token, we need to capture any **errorCodes** associated with the call; these are also captured and held in the `$content` variable.

Since the message is in XML format we use a method from the module `XML::Simple`, namely `XMLin`, in order to transform the XML data format – complete with mark up soup - to a Perlish one – that is, containing values accessible as Perl data types. As an interim step, the first variable to which the transformed data will be assigned is a reference to a Perl data structure, represented by `$ref`, so next we declare that variable (as in `my $ref`). At this point we must remember we are working with a Perl reference to the data as opposed to the data itself, and therefore to dereference the data to arrive at individual variables of use. This is a common feature for all the functions. First, however, as a final check that the content is valid, we wrap our `XMLin` function within an `eval` statement, so if there is a problem with the content (for example, if the call has failed and an XML data structure is not represented within the `$content` variable), we do not necessarily have to exit the program, but can capture the problem, as follows:

```
eval { $ref = XMLin($content) };
if ($@) {       print Dumper ($content);
                print "$@\n";
                die "login failed to retrieve valid XML"};
```

The objective is to subsequently define a Perl data structure containing the variables which we are interested in returning from the function. So this is the last part of the function. First, we set up the empty hash that we want to populate with key value pairs:

```
my %login_hash = ();
```

Then, we populate with the required key value pairs for our hash by searching the data structure represented by `$ref`. These values can be found by using the statement `print Dumper(\$ref)` (note that it is necessary to insert a backslash **** before the variable `$ref` in order to dereference it and get at the values).

```
$login_hash{sessionToken}     =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'header'}
{'sessionToken'}{'content'};
$login_hash{headererrorCode}  =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'header'}{'errorCode'}
{'content'};
$login_hash{errorCode}        =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'errorCode'}{'content'};
       return %login_hash;
       }
```

By specifying our required value using the form `$ref->{etc}{etc}{content}` we store the actual value as opposed to a reference to it, and store that within the key `{sessionToken}` to our `%login_hash`. Likewise, with the **headererrorCode** and **errorCode** keys and values.

There we have it. To use the call in separate programs we first call the library, as discussed above, then specify the arguments to the function (i.e. Function name followed by arguments within brackets) and save the output to a variable. The variable must be the required return data type from the function - in this case a hash (we list all the data types returned from the examples subsequently. Thus, to use `login` within a separate program, assign the relevant variables and make the following statement:

```
%login = login($username, $password, $productId);
```

To obtain individual values, we look in the return variable, in this case the `%login` hash, and extract the values as shown in the library (or by interrogating the output of the response variable for other values). So, for the objective of our function, to extract the sessionToken as a result of logging in, enabling further calls to be made within the program, we state:

```
$our_token = $login{sessionToken}
```

Simple enough. Although all the above may seem like a lot of work to get at the **sessionToken** alone, once each function is defined and stored away in a library it is reuseable, so we don't have to specify subroutines again (with the exception of modifying for changes in the Betfair API, of course). Fortunately, we also have a finite vocabulary to deal with in terms of services, leaving us free to concentrate on the programs which use the calls to implement a particular betting strategy – an infinitely more interesting exercise.

The arguments and outputs for all functions in the library (including `login`) are summarised in the last section of this Chapter, *Arguments and Outputs for Example Functions*. Next let's review a more complicated service in similar detail, **GetMarketPricesCompressed**, and the Perl function which calls it: `get_market_prices_compressed`.

Review of the `get_market_prices_compressed` example function

Having reviewed the function for the **Login** service, the good news is that we don't have to start from scratch to understand other Perl functions in the library that call API services. The approach is a common one for all services, with the major differences being the exchange endpoint URLs that have to be called (depending on which service is required and for which markets), and the complexity of the data that is returned by a successful call. The latter can have quite an impact on the size and complexity of the function, as is the case for **`get_market_prices_compressed`**.

We are considering the **GetMarketPricesCompressed** service for our next function to highlight the difference in the complexity of the response data returned, so it can serve for other subroutines the user might create, and because it is probably the most frequently used information service across all betting strategies.

The objective of the function is to get the current market prices for all contenders on any given market event. The information returned is exactly the same as the **GetMarketPrices** service in this respect, so the data descriptions for the service apply to both calls.

The difference between the two services is purely the way the data is presented in the response from the server. A call to **GetMarketPricesCompressed** returns, as the name suggests, a compressed form of market data where all information is presented, within the XML tags, on one line; within the line, data is separated by colon **:**, semicolon **;** tilde **~** and pipe symbols **|**. We show a function to call **GetMarketPricesCompressed** in preference to **GetMarketPrices** here since there is no difference in the values that can be obtained, but the condensed form of the data means we can ensure that prices are returned by this function quickest in order to make a betting decision.

One way of thinking about the data available from calling this service (or indeed any service) if employing it for the first time is in terms of the information we can obtain via the normal Betfair web interface and how that relates to the data that can be obtained from the service requested. In the case of calling **GetMarketPricesCompressed,** we can obtain the full range of market prices and corresponding volumes that are normally displayed on the standard Betfair user interface for every runner in the market - from the currently available back and lay prices, and corresponding volumes for each runner, to a further depth of the next 2 available back prices and volumes, along with the next 2 available lay prices and volumes. In other words, all the price and volume information that is available by contender as per the example screenshot in *Figure 4-2*.

5 (8)	**Red Opera** J P Spencer	**3.9** £34	**3.95** £252	**4** £499	**4.1** £11	**4.2** £109	**4.3** £227
1 (4)	**Missoula** J Quinn	**5.1** £66	**5.2** £402	**5.3** £31	**5.4** £49	**5.5** £67	**5.6** £58
3 (3)	**Plane Painter** Greg Fairley	**5.1** £77	**5.2** £349	**5.3** £809	**5.5** £158	**5.6** £158	**5.7** £482
2 (7)	**Valance** N De Souza	**9.4** £192	**9.6** £91	**9.8** £26	**10** £37	**10.5** £78	**11** £67
9 (2)	**Hows Business** F Norton	**8.6** £31	**9** £8	**9.6** £40	**10.5** £91	**11** £30	**11.5** £13
6 (6)	**Sweetheart** Jim Crowley	**15.5** £75	**16** £105	**16.5** £15	**17.5** £9	**18** £10	**18.5** £44
4 (5)	**Love Brothers** D Holland	**17** £34	**17.5** £75	**18** £20	**18.5** £3	**19** £15	**19.5** £27
7 (9)	**The Composer** D Sweeney	**17.5** £5	**18** £16	**18.5** £12	**21** £6	**22** £11	**23** £2
8 (1)	**My Legal Eagle** T Dean	**36** £23	**38** £9	**40** £2	**42** £3	**44** £3	**46** £9

Figure 4-2: Betfair Live Price and Volume data, corresponding to data returned by the GetMarketPricesCompressed API Service

In addition, as we will see when we examine the example Perl function call to **GetMarketPricesCompressed**, other information is returned by the API service that is only available in the website user interface by clicking on other hyperlinks. For example, the **GetMarketPricesCompressed** service returns the summary market data, as displayed in the header of the *Market Summary* screenshot in *Figure 4-3*, for each horse that can normally be found by clicking besides any horse's details.

Betting on: Red Opera

Total matched on this event: **£31,405**
Reduction Factor **23.2%**
Betting summary - Volume: **£10,228**
Last price matched: **4.10**

Figure 4-3: Betfair Live Market Summary Data corresponding to the GetMarketPricesCompressed API Service

The **GetMarketPricesCompressed** service also returns information about the rules of the market, including any runners which have been removed from the market (i.e. Non-runners) such as we would normally find under the "Rules" tab on the right of the market interface screen, as below:

Race card No.	Reduction factor	Withdrawal time
8. Magnum Opus	2.7%	08:42 (UKT)

Figure 4-4: Betfair Live Market Non Runner Information corresponding to the GetMarkets and GetMarketPricesCompressed Services

Mapping information between available services and the website user interface is a good way for the automatic bettor to start thinking about what services might be useful to automate their own strategies; however, the examples shown - in terms of how the user interface relates to the API service for **GetMarketPricesCompressed** - also demonstrate that the services need to be digested and treated as a vocabulary in their own right. Certain information we might associate as being of the same type in the user interface can in fact be found in the API only by calling more than one service, just as using more than one service can bring together information that is found in disparate places on the user interface. The difference when using API services is of course that

we have complete control over how and what information is available and – if we wanted to program a new user interface, for example, as opposed to automating our betting - how it is displayed.

Note, for example, that the all the data required to make the *Price/Volume over time* graphic shown in *Figure 4-4* (which represents all prices and volumes matched for any particular runner) is not available from the **GetMarketPricesCompressed** service, which rather returns the prices and volumes available to bet (either backing or laying) as opposed to what has been matched.

For that data, on matched prices and volumes (although not timestamped), we must look to the total trading history for each horse, which can be retrieved by the **GetMarketTradedVolume** service. In the same vein, we should note that the price information returned by the **GetMarketPricesCompressed** service for each contender is returned according to the contender's Betfair identification number rather than the contender's name. To retrieve the runner name, not just the Betfair selection ID, we also need to call the **GetMarkets** service. This is frequently a requirement in the context of automated betting strategies where we wish to compare our own odds, which will be stored by horse name, to the Betfair prices for those horses. Thus it is necessary to retrieve the static data (i.e. the runner name matched to the Betfair runner identification number for that contender) by using the **GetMarkets** service, prior to calling **GetMarketPricesCompressed** and matching prices according to the runner or selection ID. There are many choices as to how and when this is done (for example mapping names to identification numbers before racing begins, or dynamically when the program is run), but clearly it is necessary to know what information each service provides in the first place. We will explore common combinations for such functions in subsequent chapters describing how to setup the automatic betting framework - prior to implementing strategies in Part 3.

Code review:
So much for an overview of the **GetMarketPricesCompressed** service, and some of the services that are typically used in conjunction with it.

Let's now look at the implementation of the Perl function that calls this service, `get_market_prices_compressed`. To begin, we define our subroutine, or function, and the arguments to the function, as we did previously for the first function described, `login`.

```
sub get_market_prices_compressed
      {
  my ($sessionToken,$marketId)=@_;
```

In this case, the first argument is the session token, which will be a common argument for any service call in the library subsequent to calling the `login` function itself. This variable we must have obtained as per the previous section (as one of the return values from the **Login** service), and here supply it as the first argument to the call, as denoted by the variable `$sessionToken`. The second argument is the Betfair market identification number (`$marketId`) for the market we wish to retrieve prices for.

To obtain the value of the Betfair market ID as an argument to supply to this function, we would normally run the function to call **GetEvents** or the **GetAllMarkets** service call

prior to requesting market prices, in order to capture the Betfair identification of all the events that we are interested in. How we then go about choosing and specifying the market identification numbers in which we are interested depends on our betting strategy.

In terms of process, which **marketId** we choose depends upon the other stages in the framework that come before making any call for market information and upon betting strategy. Capturing, storing and subsequently re-using event data for daily markets is covered in the next chapter. For now, we can take it as read that there is a process of capturing events and automatically choosing the required event prior to using `get_market_prices_compressed`, and we cover this in detail later. Here, however, we will consider the function in isolation, therefore assuming we already have a market ID to supply as a second argument to call the function.

The first part of the function call itself, shown below, is similar in structure to `login`, in order to specify and make a valid request to the Betfair servers for the required information. Of course, the specific XML that must be used within the SOAP envelope for a valid request to this service is different, and includes the name of the service and parameters particular to it, as detailed in the *Betfair Sports Exchange API 6 Reference Guide*.

```
my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmln$
        <SOAP-ENV:Body>
                <m:getMarketPricesCompressed
xmlns:m="http://www.Betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <marketId>'.$marketId.'</marketId>
                        </m:request>
                </m:getMarketPricesCompressed>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
```

As per the `login` function, we also initialise the user agent to which we will pass the XML data in order to request the service.

```
        my $userAgent = LWP::UserAgent->new();
```

And again as per the `login` function, we specify our request, this time to the exchange specific server from which we can retrieve the data relating to events.

```
        my $request = HTTP::Request->new(POST =>
        'https://api.Betfair.com/exchange/v5/BFExchangeService');
        $request->header(SOAPAction =>
        '"https://api.Betfair.com/exchange/v5/BFExchangeService"');
        $request->content($xml);
        $request->content_type("text/xml; charset=utf-8");
        );
```

Note in particular the endpoint URL which is used to direct the request to the Betfair servers*: https://api.Betfair.com/exchange/v5/BFExchangeService*
This endpoint specifies the UK exchange, which means that we are calling the **GetMarketPricesCompressed** service for the all events which are taking place outside

Australia. In order to access Australian specific events with this service, we would specify a URL in substitute of wherever the above is used as follows: *https://api-au.Betfair.com/exchange/v5/BFExchangeService.*
This split in service calls between the UK exchange and the Australian exchange is as a result of changes Betfair has had to make to its business operations in order to be compliant with the Tasmanian Gaming Commission, which were introduced in API version 5. Now there are **global services** via the Global Exchange URL (such as **Login**, and non-event specific services such as checking Account balances and so forth) and **exchange specific services**, meaning any service specific to an event (e.g. getting prices or betting in an event) via the UK or AUS Exchange URL. Therefore the services available under the API are the same for both exchanges, but the exchange (and thus the events) that are accessed depend upon which exchange is specified in the Perl function calling the service.

We cover specification of the relevant exchange as a general topic for all service call functions in more detail in the last section of this chapter. For now, suffice to say there are many ways of handling this, including the ability to specify which exchange service we want to use, on a program by program basis (depending on whether we want prices on an Australian event or not). This will be up to the bettor, and their preferred set up. In the code above, the XML specifies the UK exchange, meaning the setting is for accessing prices for an event outside Australia.

Next we make our request, check the results by enclosing it within an eval statement, then convert the response from XML to a Perlish data structure, stored in the reference variable `$result`.

```
my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
#print Dumper(\$result);
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n";
die "GetMarketPricesCompressed failed to retrieve valid XML"};
```

The objective is now to define and return the values from the call to the **GetMarketPricesCompressed** service that will be useful to other programs, so we create a Perl data structure containing all the variables we need to access market and price information returned for the relevant event from this service. First, we set up the empty hash that we want to populate:

```
my %prices_hash;
```

The nature of making reference to a complex data structure means, by definition, we have to deal with referencing and dereferencing issues. Next we dereference the part of our `$result` variable which is itself a reference to the data structure we need – a hash containing the results of the response to the **GetMarketPricesCompressed** service:

```
my $response=
\%{$result-> {'soap:Body'}{'n:getMarketPricesCompressedResponse'}{'n:Result'}};
```

Now we are able to assign keys and extract values from the response which can be stored within the `%prices_hash` that will be our principal data structure returned by the function.

First we extract the new **sessionToken** that is returned in the **header** to the response (likely as not to be the same as the **sessionToken** with which we made the request, but possibly not, as we explained for the `login` function, so we want to save it). Additionally we save the variables for any errors relating to the call as well as the Betfair timestamp detailing the time (as kept by the Betfair servers) when the data was retrieved.

```
$prices_hash{'errorCode'     = $response->{'errorCode'}{'content'};
$prices_hash{'timeStamp'}    = $response->{'header'}->{'timestamp'}{'content'};
$prices_hash{'sessionToken'} = $response->{'header'}->{'sessionToken'}{'content'};
```

Capturing then making provision to use this header information in betting strategies is generally good practice. The **sessionToken** value is a necessity for longer programs which will stay hooked to the API for some time, the error messages are useful for program checks and debugging, and the timestamp is critical in monitoring and assessing the feasibility of betting strategies (e.g. if there is a significant lag between functions updating price information in a trading application, it would be better not to implement the automated strategy).

The next, sizeable, part of the code deals with parsing the string of information that is returned by **GetMarketPricesCompressed,** which represents all the market prices for each contender, and some other information besides, as in the sample string shown in the documentation for the **GetMarketPricesCompressed** function in the *API Reference Guide.*

The fact that all the information is presented in a long string is what gives the function the right to its “Compressed” suffix. To get at the market information we essentially have to “decode” the string and separate it into specific prices and volumes for each contender within the event. Fortunately, Perl excels at pattern matching and data manipulation. Essentially this sums up the rest of the work in the program: namely, sorting, parsing and storing variables from the string. We will therefore comment in summary:

Save the long string as a single variable called `$runner_price`.

```
my $runner_price =
$result->{'soap:Body'}{'n:getMarketPricesCompressedResponse'}{'n:Result'}
{'marketPrices'}{'content'};
```

Get rid of colons preceded by double backslashes (refers to a specific item of information in the reduction factor field) since colons (:) otherwise signify delimiters of information about each runner in the string

```
$runner_price=~ s/\\:/colon/g;
```

Set the count for the number of runners to zero, and create the array `@price_split`, by which we will create an array of each runner’s market info - since each runner’s info is separated by a colon (**:**) and we split the string into an array separated by colons.

```
            print $runner_price,"\n";
            $prices_hash{'noRunners'}=0;
            my @price_split=split(/:/,$runner_price);
```

Next we can see the number of values in the array by treating this array as a scalar, so the size of the list which contains all our information is saved in the `$size` variable

```
            my $size=@price_split;
      #     print $size,"\n";
```

However, the first value in the array `$price_split[0]` refers to generic market information, so let's deal with that first, parsing the string to split out all the individual values within it, which are separated by a tilde **~,** and saving that in turn to another array `@market_attributes`, from which we will subsequently extract each significant value and store in keys of `%prices_hash`**,** so such values can be accessed directly by relevant betting programs that use our function.

```
my @market_attributes=split(/\~/,$price_split[0]);
        $prices_hash{'marketId'}        =$market_attributes[0];
        $prices_hash{'currencyCode'}    =$market_attributes[1];
        $prices_hash{'delay'}           =$market_attributes[3];
        $prices_hash{'marketStatus'}    =$market_attributes[2];
        $prices_hash{'marketInfo'}      =$market_attributes[5];
        $prices_hash{'numberOfWinners'} =$market_attributes[4];
        $prices_hash{'lastRefresh'}     =$market_attributes[8];
         $prices_hash{'IsBSP'}           =$market_attributes[10];
                                           #returns a value of Y or N
```

The relevance of each of the variables can be referenced in the *Betfair Sports Exchange API 6 Reference Guide.* Note in particular that we can enquire on whether the event in question is subject to BSP (Betfair Starting Price) bets, with the attribute `prices_hash{IsBSP}`**.**

Now that market attributes are out of the way, we know that the rest of the array `@price_split` contains elements each of which represent a runner's details, so we loop through the remaining number of elements in the array with the `for` statement, dealing with each runner, `$t` and applying the full code block to each until no more runners are left.

For each runner, we split out arrays of market data about that runner to `@prices`**.** Each type of market data is separated by a pipe – in the order of `Header Information|Back Prices|Lay Prices`, relating to our original long string. For each of these elements of our array, we create further arrays splitting out the individual elements in Header, Back and Lay, each of which is separated by the tilde **~** separators used in the string for delimiting the individual fields (i.e. Values) we are interested in. Note that `$header[0]` which is used in the references to find each value, is the first value in the `@header` array and holds the selection identification number – so all values can be referenced by this in future.

```
for ( my $t =1 ; $t < $size ; $t++)
                {
   my @prices =split(/\|/,$price_split[$t]);
   my @header =split(/\~/,$prices[0]);
   my @back   =split(/\~/,$prices[1]);
   my @lay    =split(/\~/,$prices[2]);
   $prices_hash{'noRunners'}++;
```

```
   my $runnerId = $header[0];
   $prices_hash{'prices'}{$runnerId}{'orderIndex'}          =$header[1];
   $prices_hash{'prices'}{$runnerId}{'totalAmountMatched'} =$header[2];
   $prices_hash{'prices'}{$runnerId}{'lastPriceMatched'}   =$header[3];
   $prices_hash{'prices'}{$runnerId}{'asianHandicap'}      =$header[4];
   $prices_hash{'prices'}{$runnerId}{'reductionFactor'}    =$header[5];
   $prices_hash{'prices'}{$runnerId}{'vacantTrap'}         =$header[6];
   $prices_hash{'prices'}{$runnerId}{'farBSP'}             =$header[7];
   $prices_hash{'prices'}{$runnerId}{'nearBSP'}            =$header[8];
   $prices_hash{'prices'}{$runnerId}{'actualBSP'}          =$header[9];
   $prices_hash{'prices'}{$runnerId}{'backDepth'}=0;
   my $back_size=@back;
   my $lay_size =@lay;
```

Since the size of the arrays `@back` and `@lay` containing market information vary according to how many levels of depth there are specified in each array, we know that the number of elements, or size of the array indicates the available market depth information in that array. So we can use the size of both the back and lay arrays in the remainder of the function in order to split out the variables for back depth, price at that depth, and the amount available to bet (or volume) available at the depth – which is what the rest of the function does, for both back and lay prices.

```
if ( $back_size == 0 )
    {
$prices_hash{'prices'}{$runnerId}{'backDepth'}=0;
    }
if ( $back_size == 4 )
   {
$prices_hash{'prices'}{$runnerId}{'backDepth'}=1;
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'} =$back[0];
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'} = $back[1];
   }
if ( $back_size == 8 )
  {
$prices_hash{'prices'}{$runnerId}{'backDepth'}=2;
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'}=$back[0];
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'}=$back[1];
$prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'price'} =$back[4];
$prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'amountAvailable'}=$back[5];
   }
if ( $back_size == 12 )
  {
$prices_hash{'prices'}{$runnerId}{'backDepth'}=3;
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'} =$back[0];
$prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'}=$back[1];
$prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'price'}=$back[4];
$prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'amountAvailable'}=$back[5];
$prices_hash{'prices'}{$runnerId}{'back'}{'3'}{'price'}         =$back[8];
$prices_hash{'prices'}{$runnerId}{'back'}{'3'}{'amountAvailable'}=$back[9];
  }
if ( $lay_size == 0 )
 {
$prices_hash{'prices'}{$runnerId}{'layDepth'}=0;
  }
if ( $lay_size == 4 )
 {
$prices_hash{'prices'}{$runnerId}{'layDepth'}=1;
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}          =$lay[0];
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=$lay[1];
 }
if ( $lay_size == 8 )
 {
$prices_hash{'prices'}{$runnerId}{'layDepth'}=2;
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}          =$lay[0];
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=$lay[1];
$prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'price'}          =$lay[4];
```

```
$prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'amountAvailable'}=$lay[5];
 }
if ( $lay_size == 12 )
 {
$prices_hash{'prices'}{$runnerId}{'layDepth'}=3;
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}          =$lay[0];
$prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=$lay[1];
$prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'price'}          =$lay[4];
$prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'amountAvailable'}=$lay[5];
$prices_hash{'prices'}{$runnerId}{'lay'}{'3'}{'price'}          =$lay[8];
$prices_hash{'prices'}{$runnerId}{'lay'}{'3'}{'amountAvailable'}=$lay[9];
    }
 }

my $prices = 'prices';
```

Finally we return our data structure to the program for reuse

```
        return %prices_hash;
}
```

All done.

To retrieve any of the values from the `%prices_hash` within a program (assuming we have made the call to the **GetMarketPricesCompressed** and returned the hash of price values as the output from using the `get_market_prices_compressed` function), we can now use standard Perl data extraction methods applied to the hash. In principal, this is a matter of specifying the relevant key (or key sequence in the case of price information) to the data hash in order to return the value. We can hunt down "the relevant key" by inspecting the function definition above (or using `Dumper` at any point on the XML structure). So, for a couple of examples to illustrate use of the hash returned by `get_market_prices_compressed`:

1. To retrieve the current back price and store it to `$back_price1` for a particular selection (assuming we have extracted and specified the runner ID available for that selection), our statement might read:

```
$back_price1 = $prices_hash{prices}->{runnerID}->{back}->{1}->{price}
```

2. To retrieve the available lay volume at a depth of 3 (i.e. Not the volume available for the current lay price, or the next available lay price, but the last shown in the queue for any particular selection on the Betfair website interface), and then to save it in the variable `$lay_vol3`, our statement might read:

```
$lay_vol3 = $prices_hash{prices}->{runnerID}->{lay}->{3}-> {amountAvailable}
```

In the above we have presupposed that we have known Betfair runner identification numbers. However, to emulate a betting process, the task of obtaining market prices would typically begin with all runner names for a race, then proceed to "look up" the Betfair runner ID (or **selectionId**, for its proper API reference, and `$runnerId` in the example code snippet) using another service, such as **GetMarkets**.

Having saved all runner names with their corresponding Betfair ID numbers, we could set up a loop to retrieve market information for each of the runner names, using the Betfair runner ID (i.e. **selectionId**) as the key in the prices hash. However, remembering there is always more than one way to do it, it's worth mentioning that our

price data structure contains all the information we need to extract all runner IDs with prices, so we can map IDs to names at a later stage. Although we have one main `$prices_hash`, we have lots of "sub-hashes", one of which contains all the runner IDs as keys within it - accessed by `%{prices_hash{prices}}`. Thus we can simply return each runner ID within its own array, and then loop over that array to return whatever price and volume information we want for each runner ID.

```
        @runnerIds =  %{prices_hash{prices}};
        foreach my $id (@runnerIds)  {

#--------Insert statements to extract the market prices required for each runner…

        };
```

That's enough code explanation and walkthrough for now – the intention has been to show the way in which example service call functions – i.e. `login` and `get_market_prices_compressed` can be built to interact with Betfair API Services – in this case **Login** and **GetMarketPricesCompressed**.

All the code for the functions in the library is provided "as is" for the reader in Appendix 2. The essential elements have been covered in examples in this Chapter. From a uniquely Perl perspective, we should mention again the references in the bibliography remain the most valuable material with which to come up to speed on what the language is doing.

We will concentrate now on setting the library functions to work where appropriate.

The next section therefore provides a summary overview of the arguments and outputs for all functions in the library, as a quick reference to using those functions, without getting bogged down in the function detail. We also consider the different Exchanges (representing events in different countries) which each service can call, and how to implement that functionality if required.

Arguments and outputs for example functions

Table 4-2 lists the subroutines (or functions) included in the library, corresponding to API services of the same name. For each function, the arguments to be supplied to the function are given, along with the type of data structure returned by the function call. This can serve as a summary reference when using these functions.

There are fewer examples listed than all the API services available (see Appendix 3), but these are all that is needed for developing the majority of automated betting strategies. Any other services which may be used in addition are usually about improving performance and/or maintainability of code, or using the same services under different subscription levels in order to avoid throttling limits on data intensive strategies. The example functions described and used for example strategies are all available under the Free Access API (except `get_bet`, which is included as an example of a convenience function from the Full Access API, although we also show how to extract betting information using the Free Access API with `get_account_statement` in Chapter 11).

Table 4-2: Example Library Functions with Arguments and Return Data

Perl Subroutine Name	Subroutine Arguments	Data structure returned
login	$username, $password, $product Id	%login_hash
get_events	$sessionToken, $eventParentId	%events_hash
get_markets	$sessionToken, $marketId	@hashes; *@hashes consists of (\%names_hash, \%market_hash)*
get_market_prices_compressed	$sessionToken, $marketId	%prices_hash
get_detailed_market_depth	$sessionToken, $marketId, $selectionId	%depth_hash *N.B.: All market depth info in @ { depth_hash{depthArray} }*
get_market_traded_volume	$sessionToken, $marketId, $selectionId	%traded_hash *N.B.: All volume info in @{ $traded_hash{volArray} }*
place_bet	$sessionToken, $asianLineId, $betType, $marketId, $price_asked, $selectionId, $stake, $betCategoryType, $betPersistenceType, $bspLiability	%bet_hash
update_bet	$sessionToken, $betId, $newPrice, $newSize, $oldPrice, $oldSize, $persistType, $oldPersistType	%update_hash
cancel_bet	$sessionToken, $betId	%cancel_hash
get_mubets	$sessionToken, $marketId, $betId	%mubets_hash
get_market_pandl	$sessionToken, $marketId	%pandl_hash
get_account_statement	$sessionToken, $startDate, $endDate	%statement_hash
get_bet	$sessionToken, $betId	%getbet_hash
get_account_funds	$sessionToken, $startDate, $endDate	%statement_hash

For more detail on what data is returned within the data structure for each call, refer to the library itself in Appendix 2 (an up to date version of which can also be downloaded at *www.betwise.co.uk*), and in particular the code for each subroutine. The subroutines show the most commonly used variables saved to the data structure that is returned by the subroutine.

To see the full data structure and all variables described, the reference returned from the API service call within the Perl code can simply be dumped to standard output (using `Data::Dumper`, as explained in the previous section).

Exchange specific services and betting on Australian events

All the services we have mentioned and listed in the previous section are available for all Betfair markets whatever the location. However, some services, namely for Australian events or events operating under an Australian licence, require that the server which is accessed by those services is different from the UK server.

For the sake of clarification, we will paraphrase the following excerpt from the Betfair API documentation:

The Betfair sports betting API services are available only over SSL secure connections. For release 6.0 of the API, from which point Betfair Starting Prices and Bet Persistence is supported, there are three connection end-point URLs:

- *https://api.Betfair.com/global/v3/BFGlobalService*
- *https://api.Betfair.com/exchange/v5/BFExchangeService*
- *https://api-au.Betfair.com/exchange/v5/BFExchangeService*

The first URL is for requesting the API's global services. The global services are used to log in and out, administer your Betfair account and funds, and to navigate to the sports events you want to bet on.

The second URL is for requesting the API's exchange services that enable you to view and bet on UK and other (but not Australian) sports events.

The third URL is for requesting the API's exchange services that enable you to bet on Australian sports events (that is, on events that are taking place in Australia or that - for some other reason - have a betting market that operates under an Australian licence).

Therefore, as we saw in the function walkthrough for `get_market_prices_compressed`, any function requesting a service must specify the required exchange within the body of the function, specifically in the SOAP request. In the case of the default library we have used the general (i.e. Non AUS specific) URL. This means any service call will automatically apply to non-Australian markets. There are any number of choices available if we want to change this behaviour to incorporate AUS events.

For example, to alter our preferences completely we can do the equivalent of a search and replace on the library functions, substituting the AUS URL for the non-AUS URL. This will now allow access only to Australian events, so we could rename our library of functions as *`BetfairAPI6AUS.pm`*. By the same token, we could save our existing library as *`BetfairAPI6UK.pm`*, or leave it at the default of *`BetfairAPI6Examples.pm`*.

Given that the designer of an automated betting strategy will generally know in advance which exchange they wish to use (whatever the method of implementing services), using 2 different libraries is perhaps the most practical solution for those wishing to bet on Australian events using the examples provided. Subsequently, at the start of any program, we would simply have to state `use` *`BetfairAPI6ExamplesUK.pm`* or `use` *`BetfairAPI6ExamplesAUS.pm`*, but not both. A further approach is to maintain one library and add new subroutine names for each exchange specific service, such as `get_aus_market_prices_compressed`.

A more elegant solution for the code in the library is to supply an exchange parameter to each function's arguments, and specify within each function a choice of endpoint URLs to be used, depending on whether the new argument supplied contains "au" or "uk".

Or, more succinct still, we could remove the SOAP call part of every function, currently repeated within the body of each function call, and define that as a function in its own right, with its own arguments – one of which would be the exchange to be used, with the choice of endpoint URLs again defined within the function. The advantage of this last named method is that it makes for easier maintenance of the library – we are splitting out of the function any elements that are subject to future change. However, this is not implemented in the example library, since whilst it makes the library code more elegant it also means we have to explicitly define an `$exchange` variable within every program that uses these functions; moreover, it is not needed in the example strategies shown, since these use UK and Irish races.

All of this goes to show, again, there is more than one way to do it, and most questions we encounter have multiple answers – the right answer often depends on the use case and the user's own preferences. Now the options are on the table, the reader can make their own choice.

From this point, we have the technical background to implementation of functions. The important takeaway point is that we now have a library of functions available to us that should "just work", regardless of how deep into those functions the user wishes to go, so we can retrieve data from the API at will. Now we can start to implement the framework for automatic betting activity, which starts with the retrieval of event information.

Chapter 5: Betting process 101 - Daily event data

We said in Part 1 that the first stage of any betting process begins with consideration of a particular event. From two raindrops falling down a window to the World Cup Final, betting activity ultimately boils down to individual events (even when multiple events are combined in one bet) each of which are bound by their own rules.

However, for our process to be truly automated, the manual selection of events to bet defeats the object; the idea of complete automation necessitates the selection of events automatically, based on an appropriate automated betting strategy.

As with other aspects of the automated framework, there are occasions when bettors may want some manual intervention (such as picking and choosing events in the morning, then leaving the system to it), and a partially automated approach is of course possible, leaving aside any discussion as to whether it is desirable. However, we first proceed on the basis that all elements of the framework should execute perfectly without any intervention once they have been configured for any betting strategy (it is possible to remove some of the automated elements once they are all in place if requiring semi-automation). In this case the strategy should be programmatically able to select events, as we would, based on the same criteria that we would exercise. Therefore, the only viable way to automate event selection is to select event types (and, afterwards, individual events within those types) according to criteria which can only be determined by a betting strategy.

At a high level, criteria for automatically selecting events that fall within a strategy are:

- Event type (i.e. type of sport, horseracing, football etc)
- Event filter (e.g. non-handicaps, all races under one mile, premier league games)
- Event location
- Event time

A betting strategy can be wide reaching – for example, applying to all UK and Irish horseracing on any given day, which would assume that we had a betting method which could potentially consider all these events. Alternatively it could be very specific and low level, for example applying to Handicap Hurdles in Ireland in the first half of the National Hunt season.

We have a choice as to when to apply any filters to events to be used by a given betting strategy. Any filter can for example be applied after all events for a given sport have been captured and stored locally (be it horseracing or any other sport), immediately prior to the event start. To be as general and accommodating as possible for different betting strategies, it can therefore be better practice to apply filters per strategy, having first captured all events within a certain type.

For example, one strategy in a bettor's portfolio may be to bet on Irish handicap hurdles for which Ms. A.N.Other tipster specialises and publishes selections on her website immediately prior to racing (here the strategy may automatically download the website selections and bet them blindly against any qualifying races on the day, first having picked up and stored all potential qualifying events). Another strategy in the portfolio

may be betting on all weather sprint races at Wolverhampton with over 10 runners (where the bettor is applying their own algorithm for determining the value of the low draw bias and then comparing this to the prices available, either backing or laying overlaid or underlaid contenders). A third strategy in development may calculate and then apply an oddsline to all races in both the UK and Ireland.

In general, we therefore want to pick up and store the wider spectrum of given events on the day in order to accommodate the widest range of potential strategies. It doesn't cost us anything to do so, and further logic can subsequently be applied to filter out events which are not required. This presupposes that we have a betting strategy in place which has been programmed to cope with handling the event type in the first place. From a wide remit, our decision making programs will narrow down the events on which bets are actually placed.

Therefore, before programs can consider contenders within an event, or bet someone else's selections, our automatic betting process must have all the available events at its disposal, within the remit of the betting strategy being pursued. On the other hand, there is no point in capturing mixed doubles tennis events if the bettor is only interested in UK horseracing.

We will therefore discuss the general principle of capturing any type of event, provided that the events are available as Betfair markets. In keeping with the other examples, we will concentrate on automatically capturing events that enable us to pursue betting strategies for daily horseracing, with particular reference to racing in the UK.

Before we dive into the detail of capturing daily events, and discussing what we can do next to filter that event data for particular strategies, we should remember that the process of automating the capture of daily event data relies as much on our programming of the operating system as the programs that actually retrieve the data. This is also true of most other elements within our automated betting strategy.

To run the programs daily, untouched by human hand, we need to set up automated execution of our programs at specific times, typically by using or programming the scheduling facilities found in the operating system.

The scheduling process needs to cover all the programs which pull together a particular strategy, of which event capture will be the first part, and recording bets which have been executed on a particular event at a certain time will be the last part. We cover the scheduling process in Chapter 7, *Scheduling – The Key to Automation*.

Purpose of event data

Leaving aside the analysis of contenders within an event to determine what to back or lay, what are we trying to achieve with event summary data? At a minimum, we need to know the universe of events available on a daily basis that fit the remit of our betting strategy. We need to know the time and course of each event that is a potential candidate for any betting strategy, primarily so that we can run programs to

automatically schedule runner analysis, price gathering and betting activity around these times.

We have many choices over where to get event data, as discussed in Chapter 3, *Tools of the Trade - Hardware, Software and Data Sources*. Retrieving data from Betfair is only one option, but will provide us with the minimum summary we need to retrieve events for each daily racing market. Unsurprisingly, since it is not its objective, the Betfair event data does not tell us everything about an event that can be gleaned from a dedicated data source.

For example, prize money for the event would be one such data element typically available from a dedicated source. If form analysis is the driving method for a betting strategy, we can usually assume that the higher the prize money, the more that horses will have been prepared by their handlers to reproduce or exceed their best form. Similarly, an applied data filter could be the class of the race, if we only want to bet on quality racing (for much the same reasons as prize money). To apply such filters automatically requires access to comprehensive daily racecard data, and the ability to manipulate the data programmatically, as discussed in Chapter 3.

There is good reason for obtaining summary daily event data from Betfair, regardless of any other data sources used in addition, since later in the execution of any betting strategy we need to know the Betfair identification number of any given event, or market, in Betfair terms. We refer to this identification number for the market as the market ID, or **marketId** if referring literally to the XML element in the Betfair data structure that contains this number. Ultimately we can only bet automatically on events where electronic markets are available, so we need to know what those are.

We can do this by retrieving and then storing the market ID within a data structure on disk or a database management system. Subsequently, this identification number can be queried by any other program according to the time and course where the event is taking place. Looking up the Betfair market ID for any race by the time of the event and the racecourse is also necessary to compare and amalgamate Betfair information with other information indexed by time and course, from using third party recommendations to generating our own selections or oddslines from a programmable data source.

Note that all must be indexed by both time and course, since there are occasionally common racetimes at different courses on busy days' racing (usually on Saturdays and bank holidays) so that any event needs to be referenced by both, whether it is the Betfair market ID, or the data source for the oddsline.

Matching the Betfair market ID to a course and time can be achieved programmatically through service calls at any time in the day, but it makes sense to do so early in the day for all races and then preserve the data in a database.

We can save time in the program itself and the programming tasks by doing this. For example, we can avoid unnecessary service calls to the API in execution programs, since we will already have the market ID number available locally. Retrieving static data early also means there will be fewer service calls and postprocessing of the results of those calls in our program code, making maintenance of programs more efficient.

As a general rule, the more we can split tasks of certain types into their own dedicated programs, the more we can isolate any potential problems. By splitting out the capture of static data (as opposed to dynamic data such as price or other time sensitive execution tasks) we can make the rest of the betting process more efficient and run checks on the data for our betting strategies, before racing begins. Likewise, we can also extend the scope of our capture of event data to include the Betfair identification numbers for each contender (or selection ID) within the event.

However, in our examples, we do not choose to capture selection information at this stage, preferring instead to return the runners lining up for the race (i.e. identifying any non-runners). In other words, runner information is treated as dynamic as opposed to static, and retrieved prior to the race. The retrieval of each runner's identification is done as part of the betting strategy at a designated interval before the race, then runner information is used as part of the betting decision making process. This is of particular importance to an oddsline strategy, since a pre-built oddsline will need to be readjusted to account for the price impact of removing any non-runners.

There is no hard and fast rule, it is up to the particular strategy when such data is captured. We can still treat the selection ID as static data and store it for use at the same time as obtaining the event (a.k.a. market) IDs. Selection IDs for the day never change, so in that sense they are always static data. There are occasions when non-runner information may not be relevant to a betting strategy if, for example, we are betting a system selection where we have set a minimum price threshold based upon the historic performance of the system, rather than relative merits of the other contenders. Any non-runners will not affect our price, if the minimum price is determined by system performance over its history rather than a tissue created for the contenders on the day. If the selection is itself a non-runner, it will simply be returned as a "no bet" by our program.

In similar fashion to the event ID, whatever selection methods or data sources we use, at some point we will have to translate our selection to a Betfair identification number, so similar arguments apply for doing so early on. Alternatively, we can use other services such as **GetMarketPricesCompressed** to return non-runner names (as opposed to their selection IDs), and although there is more work to do to clean up the data, if using this call to match non-runners, we could also treat all selection (i.e. runner) IDs as static data. This would mean we could retrieve the IDs early and avoid a call to the **GetMarkets** service in our later execution programs.

Although as a topic it sits to one side of automating a strategy we already have, another clear purpose for capturing daily event data is to set up a pricing database that can be referenced by market ID, in order to help build future betting strategies. That is, by building up our own database of prices, we can research potential profit and loss according to Betfair prices as well as SP; we can look at optimal timing of bet execution (which can vary for different strategies, depending on whether backing favourites or longshots, for example), and backtest strategies, including trading strategies to profit from the price movements of various contenders.

Maintaining such a database is not a trivial task, due to the amount of data that has to be handled, and there are now vendors who supply historic data under a separate Betfair licence as a result. Nonetheless, the potential exists to grab price data for your own use, and if so doing, one of the database tables that needs to be created is for race specific

information, which will be used as a reference for other tables in the database, such as those containing price information, in order to present a full picture of an event.

Finally, we can also use the Betfair race description as a prefilter for those events we are interested in pursuing for our betting strategy, dropping those in which we are not, as discussed in the introduction. If not augmenting the automated process with specialist data, the Betfair event description can still fulfil some filter needs. For example, we may have decided in advance that our betting strategy will not play in maidens or national hunt flat races, where there is often little form to go on – such events can be filtered out from the Betfair event description, and only those races that pass this test can remain in the universe of events which later stages of our betting strategy will consider.

With a richer data source, there is more choice in the range of data elements we can filter out, in order to be very specific before analysing contenders within an event. We will concentrate at this point on the minimum requirements for daily event data, sticking with the idea that the first stage of the betting process is identifying suitable events and capturing Betfair static data, for the reasons already outlined. However, it should be clear that the order of these stages and the data sources used for daily events can easily be adapted by the bettor, as in the Perl trademark, *TIMTOWTDI* ("There Is More Than One Way To Do It").

Example Code for capturing Daily event data

In this section we consider a program to capture all daily horseracing markets in the UK and write them to a database. The code is commented "in line" to make it clear what each part is doing.

Creation of `races` database table and other program dependencies:

To run the script and several others in the examples for our automated framework, it is assumed that a database **`autodb`** has been created in MySQL (installing MySQL for this purpose and setting up the database is discussed in Appendix 1), where the user has appropriate permissions.

Further, within that database, and specifically for the purposes of this example, we will write to a table **`races`**, that should also be created. A script to run from the shell or command line to create the **`races`** table is described below:

Example 5-1: Script to create the races table within the autodb database (create_races_table.sql)

```
#CREATE table for each Betfair race event in the automatic betting example database

DROP TABLE IF EXISTS races;
#@ _CREATE_TABLE_
CREATE TABLE races
(
              date                  date NOT NULL DEFAULT '0000-00-00',
              country               VARCHAR(10) NOT NULL,
              course                VARCHAR(25) NOT NULL,
```

```
                time                    time NOT NULL DEFAULT '00:00:00',
                marketId                int(11) NOT NULL DEFAULT '0',
                description             VARCHAR(25) NOT NULL

);
#@ _CREATE_TABLE_
```

To run the script and create the table, assuming you do not have to specify a password to access your database (in which case that would be added under the **-p** flag), the command is:

```
> mysql -u username autodb < create_races_table.sql
```

The example database table is a simple one which can be used "as is" for the purposes of storing event information which can be used to generate queries for individual betting strategies. Such an events table could also be an integral part of an automated betting database (e.g. including historic prices and bets placed as additional tables)

The file **abbrevs**, is also used within the body of the script, and is listed in Appendix 4. The **abbrevs** file is simply a list of the abbreviations used by Betfair for racecourse names, with the full names that they correspond to. The racecourse names that appear on the Betfair website interface and in the API are always abbreviated, but if we want to look up the market ID, based upon having time and course information for an event from another source, we can only do so with the course name that corresponds to the actual course name used in other sources. Thus we want to save the full course name in our database table, **races**, for future look up from other data sources, be they other databases or files from third parties, such as ratings or oddslines. The **abbrevs** file contains a hash data structure that is "read in" and subsequently evaluated within our program example, stored in the program as **%bf_course_abbrevs**.

Code listing and commentary

Example 5-2: Capturing Daily Racing Event Data for Betfair win markets in the UK and Ireland (get_betfair_races.pl)

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee 2004-2008.  All rights reserved.                   #
# The code from this example is available under the terms of the Artistic License 2.0       #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code           #
#-------------------------------------------------------------------------------------------#

#       Objective of script:
#       Find all today's UK and IRE horseracing events from Betfair
#       Fetch all event details - BetfairID, race time, course & description of event;
#       Store details to MySQL database, also create a dbm hash for shortened details and
#       data retrieval
#       N.B..  Script requires the file "abbrevs" has also been downloaded and is
#       available, and that a database has been set up (or a dbm file can be used to store
#       race details, in this case the database #lines should be commented out).


#       prerequisite modules to run this script
use lib "/home/AEB/lib";
use BetfairAPI6Examples;
```

```
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use SOAP::Lite +trace => "all";
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";                 #productId is for Betfair's Free Access API


#       declare other variables that will be used in this script
my %login;
my $token;
my $login_error;
my $event_menu;                       #"13" is the current menu ID for horseracing events;
my @races;
my %daily_card;
my $market_name;
my $full_course_name;
my $race;
my $event_time;
my $startTime;
my $hours;
my $minutes;
my $time;
my $course;
my %races;
my %bf_course_abbrevs;

#       Evaluate a text file (which contains Betfair abbreviations for English and Irish
#       course names) as a hash

#       Creates the hash %bf_course_abbrevs which will be used later - course abbreviation
#       is the key, the full course name is value.
{
open (FILE, '/home/aeb/abbrevs');
local $/;
eval <FILE>;
close FILE;
}

#       Retrieve today's date using 'localtime' functions and convert to Betfair format
my $year = ((localtime)[5] + 1900);
my $month = ((localtime)[4] + 1);
my $day = (localtime)[3];
my $date_today = sprintf ("$year-%02d-%02d", $month, $day);

#       Open the database handle for storing today's race event details
my $dbh = DBI->connect("DBI:mysql:autodb", "username") or die ("Error:  $DBI::errstr");
#substitute your database and user credentials

#       Open a hash to be stored as a dbm file (alternate, UNIX style method of storing
and re-accessing data to using dbase)
dbmopen (%races, '/home/aeb/daily_cardID', 0644);

#       login
%login = login($username, $password, $productId);
$token = $login{sessionToken};
$login_error = $login{errorCode};

#       Get all markets in horseracing, irrespective of country and date, using GetEvents
#       API service call
my %markets_hash = get_events($token, $event_menu);
```

```
#       Use the return value from GetEvents to create an array of Betfair id numbers for
#       all events

#       Loop through the array of event ids to capture only the event details we are
#       interested in - in this case, the events are for today's date, and events which
#       are UK and Ireland WIN horseracing markets only

my $market_ref = $markets_hash{markets};

my @Betfair_keys = keys ( %{$market_ref} );

foreach my $event_id (@Betfair_keys) {
        #my $event_id is the Betfair market ID for each race

$event_id =~ s/^\s+|\s+$//g;
$market_name = $markets_hash{markets}->{$event_id}->{marketName};
$market_name =~ s/^\s+|\s+$//g;
$startTime = $markets_hash{markets}->{$event_id}->{startTime};

#       Parse the information in start time to give the date and start time of the race

my ($event_date, $start) = split (/T/, $startTime);
$event_date =~ s/^\s+|\s+$//g;
$start =~ s/^\s+|\s+$//g;
my ($event_hour, $remaining_time1) = split (/:/, $start);
my ($event_minutes, $remaining_time2) = split (/:/, $remaining_time1);
$event_hour = $event_hour + 1;          # All Betfair times are in GMT, use +1 on BST,
                                        # comment out this line if not summertime
my $event_time = "$event_hour:$event_minutes";

#       Filter out any event that is not a win only event (e.g. markets for place only,
#       forecasts, distances, betting without the favourite)
#       Capture further event details by a call to GetMarkets, giving us all the static
#       data associated with the event.

#       For our purposes any event name that is not followed by further info in
#       parantheses is a win only market for UK and Irish racing.
#       This line can of course be changed to choose whatever events are of interest, or
#       indeed to capture all of them.
#       We will choose events for today's date only by comparing today's date with the
#       event date and discarding any events which do not match


unless ($market_name =~ /\([A-Z]+|F\/C|Fav|Place|W\/o/) {

#       Call the GetMarkets service, which returns 2 hashes within an array, the first of
#       which provides more event details.

sleep 12;       #overcomes throttling limit on using get_markets with Free Access API
my @market_array = get_markets($token, $event_id);
my %event_data = %{$market_array[1]};


my $country = $event_data{countryISO3};
my $race_description = $event_data{name};
my $course = $bf_course_abbrevs{$market_name};

# May further exclude events after this point, here we exclude any that do not fit with
# today's date, e.g. for GBR only, the next line would read:
# if ($date_today eq $event_date && $country =~/GBR/) {

if ($date_today eq $event_date) {

#to see output while running script:
print "$date_today, $country, $course, $event_time, $event_id, $race_description\n";
#dbm file general usage to look up Betfair id given a course and race time as a key:
#$races{"$course,$event_time"} = $event_id;

my $sql = qq(INSERT INTO races VALUES
('$date_today', '$country', '$course', '$event_time', '$event_id', '$race_description')
);
```

```
my $query = $dbh->prepare($sql);
$query->execute;


}
}       #end foreach marketId key...
}               #end 'unless this is a WIN market on a UK or Irish horserace...'

#dbmclose(%races);     #if using dbm
```

In keeping with all the scripts we are using to call Betfair services, we first import our library of functions to call services, login to the Betfair API, and make a call to the service **GetEvents,** specifying our sport of interest as a hard coded variable within the program (in this case "13" for horseracing), so that all horseracing events – for all times (today and future), types (win, place, forecast, special and so on), and from every country within the exchange - are returned and stored in memory within the Perl data structure **`%markets_hash`**.

That's a lot of events to consider, and for our example strategies even at this stage we know that we will not use all of them (for a different strategy or research into different event types, we could amend our criteria as appropriate).

Specifically for our examples, we want to capture today's racing in the UK and Ireland – that determines our choices in terms of geography and sport. In terms of timing, we need to specify the date of events that we want to capture and filter out only those that we are interested in. On the assumption that this program is run on a daily basis, we grab the current date using Perl's **`localtime`** function and format it in the same way as the Betfair time format, so that it can be compared to the Betfair event date, and only today's events captured.

We also want to capture win markets only. When we have hitherto referred to a market we have done so homogenously, but the fact is that are many different Betfair markets related to individual sporting events. For example, a typical horserace can produce a win market, a place market, (as the two most relevant and popular markets), but also a slew of less popular markets, such as betting without the favourite, reverse forecasts, winning distances and match bets for each race.

Each market is a unique event in Betfair terms (as opposed to all markets derived from the horse race), so we must therefore be careful to specify it is the Betfair win market details that we wish to capture (by default, we will henceforth assume when we refer to a market in a horse race that it is the win market, unless otherwise specified). However, the win market is the parent of other markets in the sense that it carries the description of the race, and all other market descriptions are suffixed with qualifying words if the event in question is not the win market (e.g. Place, Wo Fav etc).
Place markets will also no doubt be the next most popular markets of interest, after the win, in automated betting strategies. In such cases, for the code in *Example 5-2*, we can alter our criteria to say "if the market contains the word place" (i.e. **`if (/Place/)`**) rather than excluding the place market – the same goes for any of the other market types currently excluded if wanting to capture those instead. In any event (no pun intended), we can effectively interpolate Place market identification numbers by capturing the win market, since the place market identification number is usually plus one, as we discuss in the next section *Running the Program.*

Any non-UK and Irish events are followed by parantheses indicating the country of origin in the event description. Since we only want to capture UK and Irish races in this example, we can therefore go with the flow and exclude anything that is followed by a suffix (meaning all other countries and non-win markets will be dropped). At this point, having selected the subset of events we are interested in, we can use a second service, **GetMarkets,** to retrieve specific details about the events we are interested in, with a view to saving these to the database, for the reasons described in the previous section.

The last part of the program applies some more filters, namely checking that the events are occurring today, before finally saving the data we want to keep and writing it to the **`races`** database table.

Running the program

The program is designed to be run prior to daily racing and automated as a daily scheduled task. Such a scheduled task can be configured once and then set to work daily by an appropriate scheduling facility or daemon, as discussed in detail for Linux systems using command line scheduling facilities in Chapter 7, *Scheduling – The Key To Automation*.

Running the program on an scheduled basis will automatically populate the database table **`races`** (and/or a hash file saved as a .dbm file, depending on whether the user chooses to uncomment and use these lines in the script). We point out the possibility to create a local *dbm* file for convenience (effectively saving our program hashes to disk on UNIX type systems). This is because creating a database table extends the applications to be used and is not strictly necessary if all we want to do is to look up the Betfair ID numbers for the days' races.

On the other hand, being able to query the database for race details is a useful feature to interrogate and provide error checking, as well as for the reasons listed previously in discussing the uses to which event data can be applied. Moreover, we can use the database table to generate useful queries that will help with our other scheduling and automation tasks.

Let's fire up MySQL from the shell (see Appendix 1 regarding setup and configuration of MySQL) and have a look at our formatted data from running the example program. For the queries that follow we use the interactive command prompt from within MySQL (represented by **`mysql>`** in the example queries), although the same query syntax can be used when we access the database programmatically (and thus automatically) by using the Perl DBI (database interface) to retrieve query results.

```
mysql> SELECT * FROM races WHERE date="2007-10-13" ORDER BY country, course, time;
```

We did not care about the order in which we stored the races for the day, so when we issue the query to look up our races, we need to be explicit about how the information is presented by using the **`ORDER BY`** statement. In this case we order the events captured first by country (so British races (GBR) will appear first), then by course, and then the time within each course - much as we would expect to view racecard data, but

in this case it is metadata relating to each event (as opposed to the runners, riders and other racecard information) as below:

date	country	course	time	marketId	description
2007-10-13	GBR	Ascot	13:10:00	20631293	5f Grp 3
2007-10-13	GBR	Ascot	13:45:00	20631295	1m4f Grp 3
2007-10-13	GBR	Ascot	14:20:00	20631297	1m4f Hcap
2007-10-13	GBR	Ascot	15:00:00	20631299	1m Grp 3
2007-10-13	GBR	Ascot	15:35:00	20631301	7f Cond Stks
2007-10-13	GBR	Ascot	16:10:00	20631303	1m2f Hcap
2007-10-13	GBR	Ascot	16:40:00	20631305	5f Hcap
2007-10-13	GBR	Bangor	14:25:00	20634384	2m1f Hcap Hrd
2007-10-13	GBR	Bangor	14:55:00	20634386	2m4f Nov Hrd
2007-10-13	GBR	Bangor	15:25:00	20634388	3m Hcap Chs
2007-10-13	GBR	Bangor	16:00:00	20634390	2m4f Nov Chs
2007-10-13	GBR	Bangor	16:35:00	20634392	2m1f Nov Hrd
2007-10-13	GBR	Bangor	17:10:00	20634394	2m1f Hcap Chs
2007-10-13	GBR	Bangor	17:40:00	20634396	2m1f NHF
2007-10-13	GBR	Chepstow	13:25:00	20633172	3m Hcap Chs
2007-10-13	GBR	Chepstow	14:00:00	20633174	2m Hcap Hrd
2007-10-13	GBR	Chepstow	14:35:00	20633176	3m Nov Chs
2007-10-13	GBR	Chepstow	15:05:00	20633178	2m4f Hcap Hrd
2007-10-13	GBR	Chepstow	15:40:00	20633180	2m Nov Hrd
2007-10-13	GBR	Chepstow	16:15:00	20633182	2m Nov Hrd
2007-10-13	GBR	Chepstow	16:45:00	20633184	2m NHF
2007-10-13	GBR	Hexham	14:15:00	20633191	2m Beg Chs
2007-10-13	GBR	Hexham	14:45:00	20633193	2m Nov Hrd
2007-10-13	GBR	Hexham	15:15:00	20633195	3m1f Class Chs
2007-10-13	GBR	Hexham	15:50:00	20633197	3m Hcap Hrd
2007-10-13	GBR	Hexham	16:25:00	20633199	2m4f Mdn Hrd
2007-10-13	GBR	Hexham	17:00:00	20633201	2m4f Hcap Chs
2007-10-13	GBR	Hexham	17:30:00	20633203	2m NHF
2007-10-13	GBR	Kempton	18:50:00	20633266	5f Hcap
2007-10-13	GBR	Kempton	19:20:00	20633268	1m2f Claim Stks
2007-10-13	GBR	Kempton	19:50:00	20633270	7f Mdn Stks
2007-10-13	GBR	Kempton	20:20:00	20633272	6f Mdn Stks
2007-10-13	GBR	Kempton	20:50:00	20633274	1m4f Hcap
2007-10-13	GBR	Kempton	21:20:00	20633276	1m Hcap
2007-10-13	GBR	York	14:10:00	20634370	1m1f Hcap
2007-10-13	GBR	York	14:40:00	20634372	2m2f Hcap
2007-10-13	GBR	York	15:10:00	20634374	6f Listed
2007-10-13	GBR	York	15:45:00	20634376	6f Hcap
2007-10-13	GBR	York	16:20:00	20634378	1m Mdn Stks
2007-10-13	GBR	York	16:50:00	20634380	1m2f Hcap
2007-10-13	GBR	York	17:20:00	20634382	1m6f Hcap
2007-10-13	IRL	Fairyhouse	14:30:00	20633505	2m2f Mdn Hrd
2007-10-13	IRL	Fairyhouse	15:00:00	20633507	2m2f Hcap Hrd
2007-10-13	IRL	Fairyhouse	15:30:00	20633509	2m5f Beg Chs
2007-10-13	IRL	Fairyhouse	16:00:00	20633511	3m Beg Chs
2007-10-13	IRL	Fairyhouse	16:30:00	20633513	2m5f Hcap Chs
2007-10-13	IRL	Fairyhouse	17:05:00	20633515	2m INHF
2007-10-13	IRL	Fairyhouse	17:35:00	20633517	2m INHF

Figure 5-1: Daily Horseracing Events, automatically captured, shown by country, course and time

Betfair horseracing place markets are generally referenced by convention with an identification of +1 on the Betfair market ID, with the exception of all weather courses, which can be -1 to the Betfair win market identification number. This can be a quick solution to accessing a place market within a program, saving *"Betfair win marketId+1"* to a new variable to represent the place market. However, given that such conventions can change or may not always be observed, the only 100% surefire solution is to capture place markets explicitly if wanting to use them in automated strategies.

In the table above, we captured Irish events (just to show we could), though we are not using Irish events for example betting strategies on this day. If we just want UK races as a subset of the races available we can say:

```
mysql> SELECT * FROM races WHERE date="2007-10-06" AND country = "GBR" ORDER BY  course, time;
```

Similarly, if we want to bet only on handicaps in the UK, excluding all other races, we can use the MySQL syntax for simple pattern matching on text, to find any races today which contain "Hcap" anywhere in the description, as in:

```
mysql> SELECT * FROM races WHERE date="2007-10-13" AND description LIKE "%Hcap%" AND country = "GBR" ORDER BY course, time;
```

Let's have a look at the results of the last query, which cuts down the qualifying events by more than half.

date	country	course	time	marketId	description
2007-10-13	GBR	Ascot	14:20:00	20631297	1m4f Hcap
2007-10-13	GBR	Ascot	16:10:00	20631303	1m2f Hcap
2007-10-13	GBR	Ascot	16:40:00	20631305	5f Hcap
2007-10-13	GBR	Bangor	14:25:00	20634384	2m1f Hcap Hrd
2007-10-13	GBR	Bangor	15:25:00	20634388	3m Hcap Chs
2007-10-13	GBR	Bangor	17:10:00	20634394	2m1f Hcap Chs
2007-10-13	GBR	Chepstow	13:25:00	20633172	3m Hcap Chs
2007-10-13	GBR	Chepstow	14:00:00	20633174	2m Hcap Hrd
2007-10-13	GBR	Chepstow	15:05:00	20633178	2m4f Hcap Hrd
2007-10-13	GBR	Hexham	15:50:00	20633197	3m Hcap Hrd
2007-10-13	GBR	Hexham	17:00:00	20633201	2m4f Hcap Chs
2007-10-13	GBR	Kempton	18:50:00	20633266	5f Hcap
2007-10-13	GBR	Kempton	20:50:00	20633274	1m4f Hcap
2007-10-13	GBR	Kempton	21:20:00	20633276	1m Hcap
2007-10-13	GBR	York	14:10:00	20634370	1m1f Hcap
2007-10-13	GBR	York	14:40:00	20634372	2m2f Hcap
2007-10-13	GBR	York	15:45:00	20634376	6f Hcap
2007-10-13	GBR	York	16:50:00	20634380	1m2f Hcap
2007-10-13	GBR	York	17:20:00	20634382	1m6f Hcap

Figure 5-2: Daily Horseracing Events, queried to return UK Handicaps only

As it happens, the last query contains some interesting high class (and valuable) handicaps at Ascot and York. If the betting strategy was to bet races over a certain value or class, we would therefore need to augment this data with more daily racecard details from a richer programmatic data source, enabling us to filter by prize money, class of event etc. For now, this data provides a useful summary for betting framework purposes.

These examples should give a taster of the type of data we are working with and how it is presented (for those unfamiliar with the data source) and the way in which we can work with it (for those unfamiliar with databases) if going down this route.

The results we have looked at so far from MySQL have been returned by issuing interactive queries from within MySQL. However, the really useful thing about using MySQL in conjunction with Perl and its Database Interface (*DBI*) module (as we discussed in Chapter 3, *Tools of the Trade - Hardware, Software and Data Sources*) is that we have reliable programmatic access to the database.

In other words, we can run any of the interactive queries shown above by using a Perl program to do so on an automated basis, and then use the query results within our programs. We can also run various MySQL commands from the Linux shell prompt, meaning that we can save a MySQL query statement within a file, make the file executable, and schedule that to run at a time of our choosing. This gives us many choices when it comes to scheduling queries on our data to run automatically.

Later, in order to schedule programs around racetimes, we build a "ladder" of racetimes by querying the **`races`** database table and saving the results to a file. For this query, the results we are really interested in are all daily race times, so that further programs can be scheduled as a consequence of the race time. Therefore, the race times, sorted in the correct order, in addition to the course (since sometimes a time can be common to more than one race) is all we return from the query. For now, to see these results from our interactive mysql prompt (later we shall show programmatic queries from Perl), we can say:

```
mysql> SELECT time, course FROM races WHERE date="2007-10-13" AND country = "GBR" ORDER
BY time;
```

We discuss in Chapter 7, *Scheduling – The Key to Automation*, how to programmatically generate and use such a query to schedule betting programs.

Extending the scope

There are many ways the program can be extended in scope, as with all the programs in the framework; here we point out a few of the options.

The first point of note is more of an adaptation as opposed to extension. **GetEvents** is a global call, so all horseracing events are returned from around the world by using the identifier for horseracing. If we wanted to look at specific geographies or market types to those chosen for this example, we could take a different parsing route with our event details. For example, instead of using the **`unless`** statement to eliminate events we do not want, we could explicitly say (in pseudocode terms): "**`if`** the **`$market_name`** variable contains an text identifier for events we want, such as Australian races, **`then`** use the **GetMarkets** service to acquire those event details".

Note that **GetMarkets** is a local exchange service as opposed to a global service, so any corresponding **`get_markets`** function in Perl would need to be Australian (AU) specific if calling an AU event, or the function would need to be modified to accept an argument for the exchange required. We can also change the timing and type of markets returned by the program, bearing in mind we don't really have to do anything different to return place markets since they are generally *win market identification number* ***+1*** and can thus be accessed by using our existing win market data. If we

chose a different event menu identifier, we could pursue the same options above in terms of filtering the events data criteria, but with a different sport.

Capturing static data for all runners in race: We mentioned in the introduction to this chapter the option of retrieving and storing static data for each runner ID within a particular market, at this stage of the framework.

In fact the way in which the **get_markets** function call is implemented makes it easy to extend *get_betfair_races.pl* to do this. Our **get_markets** implementation returns two hashes, the first of which is a hash containing all current runners in the race and the Betfair **selectionId** number of each – where the hash key is the runner name and the hash value is the runner's Betfair ID number. So we can get the runner information for every event at the same time as we get the event details – since it simply involves bringing into play the second hash that is returned by the example Perl function, **get_markets**, that has already been called. In terms of *Example 5-2*, we already returned the second hash and saved it as **%event_data** to get the event details above, as in:

```
my @market_array = get_markets($token, $event_id);
my %event_data = %{$market_array[1]};
```

So to get the static data for runners from the first hash and save it as **%runner_data** we simply need to add:

```
my %runner_data = %{$market_array[0]};
```

Now we can extract static runner data from the hash. For the examples in the book, we also make this call when we get prices, so we will see the runner data returned using the syntax as above when we look at obtaining market information in Chapter 8. Returning runner names with the prices makes that stage more self-complete for explanation purposes, but as with Perl, there is always "more than one way to do it".

Chapter 6: Assessing Contenders

In betting on outcomes, there must be an assessment of the relative chances of contenders in any event before a betting decision can be taken. It is true that this may be an implicit as opposed to explicit stage of the framework (e.g. if taking as inputs someone else's assessment or selections), but we can still say it has formed part of the overall betting process.

In Part 1, we discussed some of these methods for assessing contenders and winning chances. It may be that most contenders in an event are practically ignored by systematic methods and all the available data attributes used in other approaches. Some approaches rely on implementing a statistical model, others on hypothetical and rule based approaches, and so on. The simple, unifying theme of all methods is to identify an edge in predicting the outcome of a sporting event.

An exception to this process is pure price and volume trading, which can happily ignore the fundamental chances of each contender in the race, thus rendering this stage of the framework redundant for trading strategies. However, there are a number of hybrid strategies which can usefully employ data related to an assessment of contenders in an event, such as identifying front running characteristics for example, for "back to lay" strategies.

Of course, it can also be that there is more than one successful outcome in the age of the exchange and the spread bet - a loser, a winner, or subset of losers or winners. Thus there is more to the objective of assessing contenders than that of identifying the winner alone.

This stage of the framework is therefore one of the most difficult to nail down with a generic example due to the variety of ways in which bettors can assess contenders.

The variety of methods used is not surprising, especially if we consider that such methods are the "secret sauce" of any particular bettor's strategy, combined with the decision making process for that strategy.

However, from an automated betting standpoint, we are primarily concerned that the methods which show the most promise can be implemented, and that our automation framework is general enough to accommodate promising methods that will arise in the future.

From this point of view, our objective in covering this stage within the framework is to demonstrate typical input, processing and useful output.

From the perspective of using the most coherent method of assessing contenders to produce the most versatile output (i.e. that can readily be adapted to different betting strategies), we will therefore look at creating and using an automated oddsline. Although many other approaches, such as identifying individual contenders with certain attributes, can be profitable, unless there is a representation of the probability of that contender winning, expressed as a percentage chance, odds to one, or decimal odds,

there is no final representation of how a contender has been assessed available to other parts of the framework.

In particular, we are concerned with the characteristics of an oddsline and the logistics for using one within subsequent parts of the framework, be it to back or lay, make a book, or dutch the field.

Within the context of producing oddslines, there are again many different approaches possible, even with exactly the same input data, although the output will consistently give us a ranking, or predicted finishing order, and a probability of winning.

For our example in the next section, we use the Postdata service from the Racing Post. The source is chosen due to being freely accessible, rather than as an example of best practice or predictive significance, although it also has merits for use in an oddsline context.

Indeed, for the automatic bettor to generate oddslines themselves, best practice would suggest using a form database where strategies can be devised and tested using the bettor's own methods. An alternative oddsline generated using an automated database is also reviewed and compared against the current example in Part 3.

In general, the subject of improving and pricing oddslines is also revisited when it comes to testing and improving strategies, including the Postdata oddsline generated in this Chapter (which is tested live) in Part 3 of the book.

In the meantime, we will start the process off by generating the basic oddsline that will be tested further from the ground up.

Generating an example oddsline using Racing Post Postdata

In this section we discuss creating an example oddsline from well known predictors that already represent a summary of each horse's comparative chances, as opposed to creating the predictors from the raw data using our own database. We are particularly interested in automating output from the oddsline that can be used within the context of the framework as a whole and can therefore be representative of any oddsline. As a method for generating oddslines, this is a starting prototype designed to show the way in which an oddsline can generally be used within the context of an automated betting strategy. A comparison with more mature oddslines is discussed in Part 3, *Automated Strategies in Practice*.

Background to the Postdata Oddsline

We reuse the variables shown in the Racing Post Postdata table, and transform these into a simple scoring system. This information is accessible in tabular format on the Racing Post website, at *www.racingpost.co.uk*, and also published in the newspaper for each race on a daily basis.

The variables assessed within the Postdata service are:

- Ability
- Recent form
- Suitability to going
- Suitability of distance
- Suitability of course
- Draw (if flat race)
- Current trainer form
- Trainer's first time out record (if debutante)
- Group Entry

The strength and weakness of each contender is assessed against each significant factor which is deemed to be predictive of the requirements for winning that particular race. Each contender is listed in the rows, and the significant variables in columns, as shown in *Figure 6-1* below, for the runners in The Hungerford Stakes, a Group 2 race at Newbury run on 18th August 2007.

HORSE	TRAINER FORM	GOING G	DIST 7.0F	COURSE	DRAW	ABILITY	RECENT FORM	GROUP ENTRY
Per Incanto	✓	✓	✓	?	-	✓✓	?	G1
Stronghold	✓	✓	✓	✓	-	✓✓✓	?	
Silver Touch	✓	✓	✓	✓	-	✓✓✓	✓✓	G1
Caradak	✓	✓	✓	✓	-	✓✓	?	G1
Wake Up Maggie	✓	✓	✓	?	-	✓✓	✓✓	G1
Dark Islander	X	✓	✓	?	-	✓✓	✓	G1
Red Evie	✓	✓	✓	✓✓	-	✓✓	✓	G1
Welsh Emperor	✓	✓	✓	✓	-	✓✓	✓	G1
Assertive	✓	✓	✓	✓	-	✓✓✓	✓	
Dubai´s Touch	✓	✓	✓	✓	-	✓✓✓	✓✓	G1
Beckermet	✓	✓	✓	✓	-	✓	✓	

Figure 6-1: Example Racing Post Postdata Table

Postdata ranks each horse against the predictive variables in terms of ticks and crosses, in the range of a cross (out of form), a question mark (unknown form), and 1 to 3 ticks as a measure of strength, depending on the variable.

We can very quickly build up a useful scoring mechanism from this information by assigning a points system to replace the ticks and crosses, converting each tick to one point, any question mark to 0 points, and a cross to a negative (the presence of a group entry might also be converted to 1 point, or 0 for none). Adding up all the points in each row, or in other words scoring all the variables, produces a score per horse. This is a manually intensive task, which we automate for the purposes of this example, but could just as easily be done from a form database.

Once we have converted each ranking to a score we can reuse each score in order to produce a ranking of comparative chances within any UK race, which can be converted easily to a basic oddsline for each contender as follows:

```
Decimal odds for each horse = 1/(Score per horse/ Sum of all scores)
```

Postdata is of course based on the Racing Post's own choice of variables and its own scoring system for each variable. However, the idea of scoring runners according to strengths and weaknesses in order to assess chance has been well used by bookies and punters for some time. Clearly we can extend or reduce the factors used to incorporate other data using our own form database, or recalculate the values from the ground up, since here we simply use the weightings for the value of each variable "as is". The question is how to build up the most accurate prediction by deciding exactly what combination of variables to use and the relative importance which should be assigned to each.

Furthermore, whatever variables are finally used, this oddsline will simply be a working hypothesis, since no adaptation of the scoring mechanism (which produces a tight numerical range) or testing against historic results has been done at this point. We will explore this further in Part 3,

For now, there is a lot to recommend the approach as a working hypothesis, as we will discuss during the course of the next section. However, it is not suggested as a substitute to the bettor developing their own "secret sauce", using original data sources.

Automating the oddsline

For this example, we use Perl scripts to log into the Racing Post website and download the required data. The Postdata table provides a convenient shortcut for building an example oddsline by repurposing summary statistics and metadata to create oddsline output, as opposed to building and applying a model from the raw data up.

Since website data is subject to change, all the caveats mentioned in Part 1, relating to the brittle nature of screen scraping, apply here. The website may change and break a script at any time (although the Racing Post layout has been stable for some time), and the policy on retrieving data automatically can be ambiguous, although permission was obtained for the purposes of generating the Racing Post examples in this book. Since we want to show how to create an oddsline for use elsewhere rather than how to capture Racing Post proprietary data from the website, we do not list the complete screen scraping script that captures the Postdata table automatically, rather we discuss the key elements and list the script which turns the data into an useable oddsline.

The automated screen scraping script was run for the duration of the trial shown in Part 3. It is a sympathetic scraping script, in that it takes all of 20 seconds to run per day and saves its results to local files which are clobbered (i.e. replaced) by each subsequent day's files, so that a permanent record is not kept offline. The principle components use Perl's **`LWP`** and **`Mechanize`** modules, which are generally the way to go (if using Perl) for any screen scraping application, since we can automatically navigate the site by "looking" (i.e. using text processing) for the links we are interested in. To do that, we must have such search terms – being the names of courses, for

example - available to the screen scraping program. Daily course names can be found on the website and parsed by the scraping application or retrieved from other sources such as the **races** table in the **autodb** database as shown in the examples in the previous chapter. Further reading on building sympathetic screen scraping and spidering scripts using the Perl modules **LWP** and **Mechanize** is listed in the Bibliography.

Those notes aside, let's continue with our example. The objective is to capture the Postdata table for each race and then generate an automatic oddsline. The Postdata table also includes the Topspeed, Official Rating and the Racing Post Rating for each horse, elements we can add to the mix. To do this we schedule our screen scraping script to run automatically each morning (further details on the mechanics of scheduling this can be found in Chapter 7).

Purely in terms of number of pages requested, being one for each race, this exercise involves no more or less interaction with *www.racingpost.co.uk* than we might have expected on a typical morning's manual browsing, although since we are following no further links into any form, it's probably fewer. By the same token, since no time is required in analysis (or counting ticks and crosses), there is a huge saving on time spent in front of the computer.

After we have captured the data for each race, we do some text processing on the returned data in memory, and extensive data manipulation to arrange all the information from the table in the correct order for each horse.

The "value added" piece of this exercise from a content point of view is in transforming all the ticks and crosses which normally constitute each Postdata variable (which can also be seen as metadata) into numeric values or scores for each Postdata variable, for each horse. Each score is represented electronically in the form of *tick.gif* images (representing one tick each), crosses (X) for negative scores and question marks(?). Our preferred scoring system will involve converting the various symbols to a numerical rating as shown in the following text parsing excerpt, together with explanatory notes, from the code:

```
s/^X\s/-1\n/g;                                          #replace X with "-1"
$_ =~ s/.*tick\.gif.*tick\.gif.*tick\.gif.*/3\n/g;      #replace x3 ticks with "3"
$_ =~ s/.*tick\.gif.*tick\.gif.*/2\n/g;                 #replace x2 ticks with "2"
$_ =~ s/.*tick\.gif.*/1\n/g;                            #replace x1 ticks with "1"
s/^-\s/0\n/g;                                           #dashes go to zero
s/^\?\s/0\n/g;                                          #question marks go to zero
```

Finally, we add up all the Postdata ratings for each variable, subtract any negative scores, and represent this as an overall numerical rating for the horse.

The resulting scoring band is quite narrow (we can change this when we test and improve the oddsline, but for now we translate everything verbatim), so we capture and use other variables in the Postdata table for extra granularity.

In this case, since we format topspeed, official ratings and RPR (Racing Post private handicap ratings) also, we also write these, together with the overall Postdata score, to a local file.

Figure 6-2 is an excerpt of the resulting file, showing the start of the file from all UK race meetings on the 13th October, being Ascot, Bangor, Hexham, Chepstow, York and Kempton, with the first two meetings from Ascot shown below (the full files are available at *www.betwise.co.uk*)

```
Ascot, 1:10, Spirit Of Sharjah,8, 123, 121, 110
Ascot, 1:10, Captain Gerrard,5, 121, 119, 108
Ascot, 1:10, Cute Ass,7, 120, 118, 105
Ascot, 1:10, Reel Gift,6, 120, 101, 97
Ascot, 1:10, Cake,7, 116, 114, 100
Ascot, 1:10, Hitchens,4, 114, 104, 99
Ascot, 1:10, Littlemisssunshine,2, 114, 112, 98
Ascot, 1:10, Dubai Princess,2, 110, 109, 87
Ascot, 1:10, Carleton,7, 108, 106, 96
Ascot, 1:10, Hammadi,5, 106, 100, 93
Ascot, 1:10, Lindoro,2, 104, 100, 89
Ascot, 1:10, Thunder Bay,7, 103, 101, 88
Ascot, 1:10, Your Pleasure,0, 93, ,
NEXT RACE
Ascot, 1:45, Rising Cross,4, 126, 102, 105
Ascot, 1:45, Brisk Breeze,9, 126, 123, 108
Ascot, 1:45, Queen's Best,8, 121, 113, 109
Ascot, 1:45, Samira Gold,5, 121, 108, 106
Ascot, 1:45, Trick Or Treat,9, 120, 119, 101
Ascot, 1:45, Athenian Way,4, 119, 116, 97
Ascot, 1:45, Kayah,5, 118, 117, 100
Ascot, 1:45, Dash To The Front,4, 114, 72, 97
Ascot, 1:45, Satulagi,5, 114, 108, 94
Ascot, 1:45, Winter Sunrise,6, 114, 103, 98
Ascot, 1:45, Party,3, 111, 76, 100
Ascot, 1:45, Loulwa,4, 98, 81, 82
```

Figure 6-2: Excerpt of data file showing repurposed Postdata variables

Each row in the file now refers to the following:

- Racecourse,
- Racetime,
- Horse name (as it appears in The Racing Post),
- Aggregated Postdata score (i.e. the sum of all the variables, as described above),
- Racing Post Rating,
- Racing Post Topspeed Rating
- Official rating

We are gradually working towards representing each horse's chances based upon a summary of its relative chances from a starting point where the data is already at summary level for each horse, as opposed to a process which uses raw data.

Within these concentrated variables, it is also worth noting that the data is race specific rather than general – which again is a few levels above using raw data alone. Therefore, the data are of little general use outside the context of the race in question - so that, for example, we cannot compare total Postdata scores across different races on the basis of their scores – each score is relative to the race conditions. For the purposes of a race specific oddsline, this is all as it should be, since we are interested in the chances of contenders relative to each other, rather than relative to an absolute

value. Anyone who bets in Class 7 races at one of the evening All Weather meetings will understand the emphasis.

Now all we have to do is turn these variables into a prototype oddsline. The oddsline, whilst a prototype in terms of testing, and whilst at this stage we will make no claim to its predictive ability, will still be of a format that makes it suitable as an example input for the rest of the betting framework – in other words it will be of the same format as a well tested oddsline.

For our initial take on pricing up the race, we will therefore make a literal interpretation of the Postdata scores, and assume that the probability of any contender winning the race is represented by the sum of all the scores. Therefore, we also assume that each contender's individual chance of winning is represented by its percentage score of the total Postdata score.

Additionally, since the band for all scores is quite tight - and more than one contender will end up on the same score - we will also rank any contenders with the same Postdata score by their best ever topspeed rating as a means of splitting them. The idea of an oddsline is concurrent with the idea of a creating a ranking or predicted finishing order, with the expectation being that our shortest priced contender has the best chance of finishing first, down to the longest priced contender.

If the range of prices produced by the oddsline creates a number of duplicates, producing a ranking order in addition to tissue prices means that when we later decide to base betting strategies around the oddsline, we also have a means of determining cut off points for the number of runners we will consider. (Ultimately, such additional data could also be used to improve the numerical range of the oddsline itself, creating a more precise superiority ranking).

The purpose of our script, then, is relatively straightforward. We must read in the data for every race, as above, produce prices based on Postdata scores for each contender within each race, and finally present all the contenders in ranked, anticipated finishing order.

Our second script, taking the concentrated variables we showed earlier, will produce an oddsline for each race from the raw data file. It will print out the bare bones needed by subsequent betting programs, so the course, time, horse name, and tissue price or odds.

Example 6-1: Creating an oddsline from summary data

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#

#This script converts summary data to tissue prices for each horse and creates a standard
#oddsline format expected by example programs that act upon the oddsline, standardising:
# a) Time formats
# b) Course names (eg. Stratford On Avon becomes Stratford) and most important,
# c) Oddsline tissue prices
```

```
open (IN, '/home/aeb/final_scores');
open (OUT, '>/home/aeb/formatted_oddsline');

while (<IN>) {

unless (/NEXT/) {($course, $time, $horse, $pdscore, $PM, $TS, $OR) = split (/,/);

$time =~ s/^\s+|\s+$//g;
$time =~ s/:/\./g;
$time =~ s/^\s+|\s+$//g;

$horse =~ s/^\s+|\s+$//g;
$course =~ s/^\s+|\s+$//g;
$pdscore =~ s/^\s+|\s+$//g;
$PM =~ s/^\s+|\s+$//g;
$TS =~ s/^\s+|\s+$//g;
$OR =~ s/^\s+|\s+$//g;

#insert rules for converting input course names to expected oddsline standards here
if ($course =~ /Bangor/) {$course = "Bangor"};
if ($course =~ /Stratford/) {$course = "Stratford"};
if ($course =~ /Newton/) {$course = "Newton_Abbot"};
if ($course =~ /Leighs/) {$course = "Great_Leighs"};
if ($course =~ /Rasen/) {$course = "Market_Rasen"};

$scores{$horse} = $pdscore;
$TS{$horse} = $TS;

#$PM{$horse} = $PM;
#$OR{$horse} = $OR;

        }       #end unless NEXT

######

if (/NEXT/) {                                           #Print out everything and reset
everything

#print OUT "NEXT\n";
                foreach $horse (sort byscore_then_speed keys %scores) {
                        push (@scores_only, $scores{$horse});
                        push (@horses, $horse);
        };      #end foreach

## adjustment all ratings to get odds based on ratings

$lowest_rating=$scores_only[-1];
$adjustment = 1 - $lowest_rating;
        foreach $item (@scores_only) {
                if ($lowest_rating < 1) {
                $adjusted_score = $item+$adjustment;
                push (@adjusted_scores, $adjusted_score)}
else
                {push (@adjusted_scores, $item)};

}

map {$total_score += $_} @adjusted_scores;

#       Now we have @adjusted_scores holding all ratings rebased to lowest value as 1,
#       plus $total_score for each.

$count = 0;
foreach $final_score (@adjusted_scores) {

$percent_chance = ($final_score/$total_score)*100;
$decimal_odds = 100/$percent_chance;
$decimal_odds = sprintf("%.2f", $decimal_odds);

$contender = $horses[$count];
```

```
$contender_TS = $TS{"$contender"};

$contender_score=$scores_only[$count];

print OUT "$time, $course, $contender, $decimal_odds\n";

$count++;
}

%scores = ();
%TS = ();
@scores_only = ();
@speeds = ();
@horses = ();
$final_score = ();
$contender_score = ();
$contender_TS = ();
$lowest_rating = ();
$decimal_odds = ();
$percent_chance = ();
$total_score = ();
$adjustment = ();
@adjusted_scores = ();
$total_score= ();
$contender = ();
$count = ();
$item = ();
$adjusted_score = ();

}
}
sub byscore_then_speed {                  #This sub to rank toprated Postdata, then speed
        $scores{$b} <=> $scores{$a}
                    ||
        $TS{$b} <=> $TS{$a}
        }  #end 2nd sub
```

Code Walkthrough for oddsline format

First, we read in all our variables from the appropriate file and open a file to write out to. Every time we encounter NEXT in the input file we know we have to consider a new race. If we don't encounter NEXT, we need to save all the variables for the race, before processing them. Since in this case we are at the final stage of preparation prior to an automated betting program picking up this file and using the horse names, tissue prices, course and time information to automatically bet, we must pay particular attention to ensuring all the variables are correctly prepared, stripping whitespace that may exist, correcting formats for times, courses and so on.

Exactly how the automatic bettor decides to do this will depend on the conventions they wish to use. Typically we will be comparing one set of horse names with another (in this case the Racing Post's and then subsequently comparing these with Betfair's when it comes to formatting the programs for betting), and the same with course names. There are always niggling differences in working with more than one data source - this program, or when we capture events, is the place to iron these out.

A case in point is "Bangor-On-Dee", since Bangor-On-Dee is the current course name in use by the Racing Post. Bangor racecourse is indeed situated by the River Dee, but in most sources, including in Betfair listings, it is simply referred to as "Bangor". The same with "Stratford-Upon-Avon" and Stratford. Failing to deal with such discrepancies can

and does result in missing betting opportunities on whole meetings, where a program fails in matching course names from different sources – so we need to test for it.

```
while (<IN>)   {

unless (/NEXT/) {($course, $time, $horse, $pdscore, $PM, $TS, $OR) = split (/,/);

$time =~ s/^\s+|\s+$//g;
$time =~ s/:/\./g;
$time =~ s/^\s+|\s+$//g;

$horse =~ s/^\s+|\s+$//g;
$course =~ s/^\s+|\s+$//g;
$pdscore =~ s/^\s+|\s+$//g;
$PM =~ s/^\s+|\s+$//g;
$TS =~ s/^\s+|\s+$//g;
$OR =~ s/^\s+|\s+$//g;

if ($course =~ /Bangor/) {$course = "Bangor"};
if ($course =~ /Stratford/) {$course = "Stratford"};
if ($course =~ /Newton/) {$course = "Newton_Abbot"};
if ($course =~ /Leighs/) {$course = "Great_Leighs"};
if ($course =~ /Rasen/) {$course = "Market_Rasen"};
```

There are any number of places in the automated set up where we can change names from one source to a common convention. For example, when reading in our course name abbreviations from Betfair in the previous chapter, we used an abbreviations file to map abbreviations to full course names.

Next we create some hashes for data we will use subsequently, for ease of access:

```
$scores{$horse} = $pdscore;
$TS{$horse} = $TS;
```

Remembering that there are some others variables we are currently discarding but can use in future in order to mix things up a little:

```
#$PM{$horse} = $PM;
#$OR{$horse} = $OR;

        }       #end unless NEXT

if (/NEXT/) {                           #Print out everything and reset everything
```

Now it's time to create the oddsline. Since we have hit the NEXT marker in the file it's time for our next race, but first we have to deal with the one whose data we have just processed. Given we are reading the whole file in line by line we can achieve all we need to – i.e. determining the oddsline for the race we have just dealt with and writing it out to a file - before we hit the first line of the next race.

The few lines below will rank each horse in the race from the highest scoring horse (in aggregated Postdata terms) right down to the lowest.

We preserve that order for each horse in the list or array **`@horses`** and the scores for each horse to another array, **`@scores_only`**. We know that the size and order of both arrays will be the same. We split them out since we want to rebase the scores for our oddsline. We know as long as we preserve the order that we can remarry these arrays subsequently.

```
                foreach $horse (sort byscore_then_speed keys %scores) {
```

```
                        push (@scores_only, $scores{$horse});
                        push (@horses, $horse);
        };      #end foreach
```

Note in the middle of the code we used a short subroutine to sort topspeed scores, as below:

```
sub byscore_then_speed {                                    #This sub to rank toprated
                                                            #Postdata, then speed
        $scores{$b} <=> $scores{$a}
                ||
        $TS{$b} <=> $TS{$a}
        }  #end sub
```

Essentially this takes as input a hash of scores and a hash of speed ratings, each indexed by the horse's name, and sorts them for us in that order – scores first, where scores are the same, speed rating takes precedence, where horses are still ranked equal, rare though that is. We rely on Perl's default sort mechanism to let the alphabet to make a distinction in rank. (Not an ideal solution, but one which occurs more often than not in the absence of data; where data is absent, we may not wish to bet at all.)

So far we have only ranked the horses. We have made our best attempt to predict the finishing order in the race, but we have no idea of the likelihood of each contender's chances. In short we have no tissue price, so let's get to that, using the method we discussed earlier.

First, we have a small problem of negative scores to overcome. Negatives are fine for ranking one contender relative to another, but they mess up our maths when it comes to looking at percentages and converting these to relative chances of winning.

Since, in probability and odds terms, all possible outcomes are represented between the range of 0 and 1 (with 0 being a definite loser and 1 a definite winner), negative scores have no place. So we determine whether negative scores, or zero scores exist, and if so, we rebase the scores by the difference in the lowest one. (Note that whilst zero is not negative it has no place in an oddsline before a race starts, since we can agree that even the biggest "rags" have some, albeit infinitesimally small, chance if lining up for any horse race, where anything can potentially happen).

In the code below, if any score is below 1, we work out a simple adjustment factor to rebase the lowest (either negative or zero score) to 1, and simply add the same factor to the score of every contender in the race. Whilst this does not keep exactly the same relationship between scores and final probability, within a small error margin it is close enough for our prototype example, which has yet to be tested and calibrated properly.

If the lowest score is already above zero, (as in the **`else`** clause below), we simply leave the scores as they are. In either case we are a left with a final array of our ranked scores, in **`@adjusted_scores.`**

```
$lowest_rating=$scores_only[-1];
$adjustment = 1 - $lowest_rating;
        foreach $item (@scores_only) {
                if ($lowest_rating < 1) {
                $adjusted_score = $item+$adjustment;
                push (@adjusted_scores, $adjusted_score)}
else
                {push (@adjusted_scores, $item)};
```

```
}
```

We also need the total aggregated Postdata score for all runners in the race before we are in a position to calculate the percentage chance each contender has, that being a fraction of the total score. So we get that total score by adding every element of the array, as in the subsequent line:

```
map {$total_score += $_} @adjusted_scores;

# Now we have @adjusted_scores holding all ratings rebased where necessary, and total
# scores
```

Last but not least, let's loop through our scores, calculate the percentage chance of each runner and represent this as decimal odds, shown to 2 decimal places.

```
$count = 0;
foreach $final_score (@adjusted_scores) {

$percent_chance = ($final_score/$total_score)*100;
$decimal_odds = 100/$percent_chance;
$decimal_odds = sprintf("%.2f", $decimal_odds);
```

Now we extract the variables we will need to write to a file….

```
$contender = $horses[$count];
$contender_TS = $TS{"$contender"};

$contender_score=$scores_only[$count];
```

Then print them out to said oddsline file, in this case simply referred to as ***formatted_oddsline***. We refer to this file as an example of input for an oddsline strategy in Chapter 9, *Automating Betting Decisions*.)

```
print OUT "$time, $course, $contender, $decimal_odds\n";

$count++;
}
```

The remainder of the program simply closes the loop for the race we have just evaluated and proceeds to wipe out values associated with any variables so they are empty for the next race, when the oddsline process starts again.

Formatted oddsline output

By this stage we have an output file, ***formatted_oddsline***, containing a tissue price for all daily races in the UK, generated automatically. Furthermore, we can and will use the tissue prices to compare to actual market prices and determine overlays and underlays, as determined by a further betting decision making program. The file is written so that it contains the variables we will need in that program. *Figure 6-3* shows an excerpt from the output file, corresponding to the race order and data that we started with from the Racing Post Postdata tables, now repurposed, with the last column representing the all important tissue price per runner. As with other example output we have discussed, the option of writing such values to a database as opposed to a flat file is always available to us (usually depending on the future availability required of the

data), remembering that subsequent betting strategy programs would have to query the database instead of reading a flat file.

```
1.10, Ascot, Spirit Of Sharjah, 8.33
1.10, Ascot, Cute Ass, 9.37
1.10, Ascot, Cake, 9.37
1.10, Ascot, Carleton, 9.37
1.10, Ascot, Thunder Bay, 9.37
1.10, Ascot, Reel Gift, 10.7
1.10, Ascot, Captain Gerrard, 12.5
1.10, Ascot, Hammadi, 12.5
1.10, Ascot, Hitchens, 15.0
1.10, Ascot, Littlemisssunshine, 25.0
1.10, Ascot, Dubai Princess, 25.0
1.10, Ascot, Lindoro, 25.0
1.10, Ascot, Your Pleasure, 75.0
1.45, Ascot, Brisk Breeze, 7.33
1.45, Ascot, Trick Or Treat, 7.33
1.45, Ascot, Queen's Best, 8.25
1.45, Ascot, Winter Sunrise, 11.0
1.45, Ascot, Kayah, 13.2
1.45, Ascot, Satulagi, 13.2
1.45, Ascot, Samira Gold, 13.2
1.45, Ascot, Athenian Way, 16.5
1.45, Ascot, Rising Cross, 16.5
1.45, Ascot, Loulwa, 16.5
1.45, Ascot, Dash To The Front, 16.5
1.45, Ascot, Party, 22.0
2.20, Ascot, Ladies Best, 13.9
2.20, Ascot, Dubai Twilight, 15.4
2.20, Ascot, Zaif, 15.4
2.20, Ascot, Akarem, 15.4
2.20, Ascot, Font, 15.4
2.20, Ascot, Peruvian Prince, 17.3
2.20, Ascot, Castle Howard, 17.3
2.20, Ascot, Red Gala, 17.3
2.20, Ascot, Crossbow Creek, 19.8
2.20, Ascot, Millville, 19.8
2.20, Ascot, Strategic Mount, 19.8
2.20, Ascot, Pevensey, 19.8
2.20, Ascot, Tropical Strait, 19.8
2.20, Ascot, St Savarin, 19.8
2.20, Ascot, Players Please, 23.1
2.20, Ascot, Nosferatu, 23.1
2.20, Ascot, Buccellati, 23.1
2.20, Ascot, Leslingtaylor, 27.8
2.20, Ascot, Mariotto, 34.7
```

Figure 6-3: Excerpt of formatted oddsline, ready for use as a betting input file

This file format will be used as the input for our betting decision making programs later. We need to be conscious about formatting when preparing any file for the final betting program, so that such files are formatted in tandem with the decision-making programs that will take them.

In this case, we have dispensed with our “NEXT” separators between each race, simply because the betting decision making program that we show as an example looks in this file for course and time information, irrespective of how it is ordered. If there is a match on time and course, the decision making program will save the details for that contender. In this way, the whole file could be jumbled up in terms of order of each race and runner, provided each line retained its integrity.

In terms of seeing how we arrived at the all important tissue prices on which betting decisions can be made, a reformatted file generated by a different script to create a report with a few more columns will show the results of our calculation and ranking process somewhat more clearly. This file is displayed for the first few races of the day for illustration purposes in *Figure 6-4*:

Course	Time	TS	Horse	PD_new	PD_first	#run	Tissue
Ascot	1:10	121	Spirit Of Sharjah	9	8	13	8.33
Ascot	1:10	118	Cute Ass	8	7	13	9.37
Ascot	1:10	114	Cake	8	7	13	9.37
Ascot	1:10	106	Carleton	8	7	13	9.37
Ascot	1:10	101	Thunder Bay	8	7	13	9.37
Ascot	1:10	101	Reel Gift	7	6	13	10.7
Ascot	1:10	119	Captain Gerrard	6	5	13	12.5
Ascot	1:10	100	Hammadi	6	5	13	12.5
Ascot	1:10	104	Hitchens	5	4	13	15
Ascot	1:10	112	Littlemisssunshine	3	2	13	25
Ascot	1:10	109	Dubai Princess	3	2	13	25
Ascot	1:10	100	Lindoro	3	2	13	25
Ascot	1:10		Your Pleasure	1	0	13	75
NEXT							
Ascot	1:45	123	Brisk Breeze	9	9	12	7.33
Ascot	1:45	119	Trick Or Treat	9	9	12	7.33
Ascot	1:45	113	Queen's Best	8	8	12	8.25
Ascot	1:45	103	Winter Sunrise	6	6	12	11
Ascot	1:45	117	Kayah	5	5	12	13.2
Ascot	1:45	108	Satulagi	5	5	12	13.2
Ascot	1:45	108	Samira Gold	5	5	12	13.2
Ascot	1:45	116	Athenian Way	4	4	12	16.5
Ascot	1:45	102	Rising Cross	4	4	12	16.5
Ascot	1:45	81	Loulwa	4	4	12	16.5
Ascot	1:45	72	Dash To The Front	4	4	12	16.5
Ascot	1:45	76	Party	3	3	12	22
NEXT							
Ascot	2:20	113	Ladies Best	10	10	19	13.9
Ascot	2:20	112	Dubai Twilight	9	9	19	15.4
Ascot	2:20	111	Zaif	9	9	19	15.4
Ascot	2:20	110	Akarem	9	9	19	15.4
Ascot	2:20	109	Font	9	9	19	15.4
Ascot	2:20	110	Peruvian Prince	8	8	19	17.3
Ascot	2:20	108	Castle Howard	8	8	19	17.3
Ascot	2:20	103	Red Gala	8	8	19	17.3
Ascot	2:20	110	Crossbow Creek	7	7	19	19.8
Ascot	2:20	103	Millville	7	7	19	19.8
Ascot	2:20	100	Strategic Mount	7	7	19	19.8
Ascot	2:20	92	Pevensey	7	7	19	19.8
Ascot	2:20	68	Tropical Strait	7	7	19	19.8

Ascot	2:20	57	St Savarin	7	7	19	19.8
Ascot	2:20	104	Players Please	6	6	19	23.1
Ascot	2:20	101	Nosferatu	6	6	19	23.1
Ascot	2:20	99	Buccellati	6	6	19	23.1
Ascot	2:20	80	Leslingtaylor	5	5	19	27.8
Ascot	2:20	106	Mariotto	4	4	19	34.7
NEXT							
Ascot	3:00	116	Yahrab	9	9	8	5.78
Ascot	3:00	117	Ibn Khaldun	8	8	8	6.5
Ascot	3:00	112	Ramona Chase	8	8	8	6.5
Ascot	3:00	111	Naomh Geileis	7	7	8	7.43
Ascot	3:00	106	Jedediah	6	6	8	8.67
Ascot	3:00	98	Meeriss	6	6	8	8.67
Ascot	3:00	105	Paveroc	5	5	8	10.4
Ascot	3:00	92	Redolent	3	3	8	17.3

…rest of file truncated

Figure 6-4: Oddsline formatted as a report

For all races, the oddsline is calculated to a 100% book, plus or minus fractional rounding errors to 2 decimal places.

We can also see above that where the lowest Postdata score is a positive number, we leave it, and derive odds from that point, yet where it is less than 1 we rebase the scores and base an oddsline from that (as with *Your Pleasure*, who scores zero on our conversion of Postdata attributes to numbers, running in the **1:10 at Ascot**.)

This is something of a crude method, but at this point the mechanism for using the oddsline within the automated framework is the key objective. The mechanism is generic whilst the oddsline content will be determined by the individual bettor's methods. (Nonetheless, we introduce more sophisticated methods for assessing the oddsline, as well as comparing oddslines for the same races, in Part 3, *Automated Strategies in Practice*.)

Next, we look at scheduling the example oddsline to work within the context of a betting strategy, to illustrate the concept of automating the entire betting process.

Chapter 7: Scheduling – The Key to Automation

Each stage of the framework, including the stages we have already covered in retrieving event details and creating an oddsline, forms a discrete piece of the puzzle in terms of automating an entire betting strategy. However, each stage cannot succeed without applying a wider logic to the automation of the strategy as a whole. Specifically, the programs within each stage must be scheduled to run automatically and to act in concert with each other.

This chapter is therefore about that logic and the glue required to bring all the stages of the framework together to make automation possible. In terms of IT skills needed, we can refer to this in terms of scheduling tasks, although that is perhaps too narrow a definition to cover all that is required – a bettor could happily use scheduling facilities but still struggle to automate a complete strategy as required. A broader view would be that automating betting strategies first requires planning and careful consideration given to timing of each aspect of the strategy.

The scheduling concept itself is simple enough: provided your operating system enables you to state in advance a specific date and time to run a program, it can run the program for you at that time.

In practice, there is more work to automating our betting strategies than a simple use of scheduling features implies, not least that we want the computer to automatically change the times at which we want programs to run every day (depending on the betting strategy) without manual editing, and to automatically know which programs we want to run at those times. However, the essential tools are available to us in order to achieve this goal. Most important is that our computer should contain a wonderful device called a clock; still more relevant is that we can read this clock programmatically. Combined with a sensible strategy and the fact that we can automatically find out when sporting events occur, we have the ingredients needed to make progress.

Linux and Perl provide us with the utilities we need to take advantage of the computer's timepiece, from **`cron`** - which is constantly looking to see if it should be doing anything in any month, week, day, hour or minute, so we don't have to - to shell functions like **`at`** and **`sleep`**, and a host of functions to access the system clock and operating system available from Perl programs (and any other mainstream programming language), some of which we have already seen.

Whilst we concentrate on Linux for examples, it should be noted that the raw tools also exist in other operating systems to automate and schedule tasks, and the general principles are common. (For example, on Windows XP systems, the main scheduling facility is **`schtasks`**, help for which can be found at the dos command line by typing **`schtasks /?`**, and full documentation at the Microsoft website. Additionally, further control on scheduling is available at the programming level, including for Perl on Windows systems. Most programming languages also have built in functions to enable interaction with the computer clock.)

In this chapter we discuss the main utilities and functions available, and how to combine these with an understanding of betting strategies on sporting events in order to provide solutions to the problem of complete automation of those strategies.

Note that the examples in this Chapter refer to some subsequent stages of the automated framework, so readers may wish to visit this Chapter again after the other aspects of the framework have been covered in the remaining chapters of Part 2.

Let's sum up one of the trickier problems we are trying to solve in terms of using standard scheduling facilities to completely automate a strategy:

Scheduling requires planning in advance when programs or tasks are going to be executed. That's great for scheduling tasks where we know in advance the specific times that tasks have to be executed (and those times never change), but what if the tasks are dependent upon events which change daily?

This is the problem we face with sporting events. The exact times that events start and finish vary every day. As we discussed earlier, the time at which the betting market is strongest (read: most liquid, and therefore the time in which we are most likely to bet at a favourable percentage book on any selection, back or lay) is in the 10 minutes leading up to the race, and therefore we would like to schedule our betting programs to run in relation to the offtime when the market presents the greatest opportunities.

We could of course ignore the possibility of time dependent programs and set everything to run in the same way at the same time every day, placing all bets very early, for example, before any racing begins. However, doing so would mean playing the market at its most illiquid, and largely ignoring any appropriate market information that might inform a betting strategy. This applies not only to price information when the market is strongest, but also to the latest market information available for the event, such as non-runners, as well as considerations relating to adjusting our own staking according to a growing or diminishing betting bank. Moreover, execution of various strategies, including trading strategies, without revisiting the market, is somewhat redundant. To revisit the market, of course, we have to be able to schedule our programs to act in relation to the event time, which is dynamic.

Even in those cases where it is appropriate for a strategy to visit the market early, the most appropriate time for doing so is still generally determined by a relationship to the time of the race – e.g. 2 hours or 30 minutes beforehand, as opposed to a fixed daily time (for example, a late Saturday night all weather meeting will tend to have little liquidity at 10 am on a Saturday, whereas a Group 1 race occurring at a afternoon meeting on the same day will generally be liquid at that time), so we therefore need to be able to schedule based on the race time and/or the event details.

In conclusion, whenever we decide to execute a program, it is usually the case that the time for capturing information and taking critical decisions, and therefore the time for running automated programs - to retrieve prices, make a betting decision, or bet, whatever - is driven by the event or race offtime.

We have another issue, related to scheduling changing event times automatically, in that whatever time we state that we want to run a program (given that we will use the same programs applied to different races), the programs that run also have to *know* which

event that is and therefore choose the correct race details – specifically the right market and runner identification in Betfair terms - for any part of our strategy. Again, such information is specific to the runners and riders on the day, and cannot be determined any further in advance than the night before the race day.

Fortunately for us, there is a degree of regularity around racing and race planning, together with available data that enables us at least to retrieve each day's events on the day or the night before. Once we have retrieved a list of events, we can use it to drive further aspects of automation. For example, this characteristic enables us to automatically schedule betting strategy programs from a daily list of race times that we know we can retrieve automatically. Thus, whilst we cannot schedule all programs to run for the right events every day in one go (since events will vary every day), we know we can rely on that information being available to do it automatically in two stages.

Using the examples from the framework, programs such as collecting daily event data and making it persistent, as in Chapter 5, as well as creating oddslines, as in Chapter 6, can be scheduled to run at a fixed time each day (let's call this a **static schedule**) which will themselves create a schedule for other programs. Programs running in the static schedule can retrieve event details, as discussed in the previous chapter, create the daily events list from the example **`races`** database table, then take the race times and input to another program which can schedule further programs to run around those times. This will effectively create a **dynamic scheduling** facility, in that we cannot know in advance when our programs will be executed, but we can schedule them based upon different daily times, provided these are available as an input. We need to construct rules around how other elements of the strategy will respond to those times. The output of our dynamic scheduling program will be thus be a sequence of programs scheduled for the operating system to pick up and execute in relation to each race time or event.

In the case of racing, all events can be uniquely referenced by the time of the race and the course at which the race occurs. Final declarations, being the final list of runners and riders, are determined the day before racing, so anytime after that (and before our dynamic daily programs have to run) we can happily fix a point in time to schedule the program that retrieves all required daily events prior to racing and makes that data persistent (e.g. stores it within a database). Such a program was discussed Chapter 5, in *Example 5-2*, `get_betfair_races.pl`. We will look at scheduling this program to run on a daily basis when we examine the utilities available to do so in the next section of this Chapter, *Linux Scheduling Facilities.*

Once we have captured daily races, we can run our dynamic scheduling program. This will depend to some extent on our betting strategy, since we can preselect a list of events that are interesting to us, and build a dynamic schedule of programs around this list to implement the automated strategy we are interested in. For the sake of argument, let us say that every race in the day is interesting to us, and a suitable candidate for betting activity. Therefore, we need to build a dynamic schedule based upon assessing every race of the day. Before going further however, let's look at some of the scheduling utilities we have been referring to.

Linux Scheduling Facilities

This section provides a brief overview of the facilities at our disposal to schedule programs, and how they can be of use in order to automate tasks relating to betting strategies. As for any command in Linux, we should also point out that there is also extensive documentation available from the shell (e.g. by typing **`man cron, man crontab,`** or **`man at`**) and further explanation within most Linux reference manuals.

Crontab

Let's deal with the crontab facility first since it is usually synonymous with the concept of scheduling in Linux.

`Crontab` is simply a file that can be edited by each user to specify tasks (executable programs, shell commands etc) to be run at a future time for that user.

`Cron`, derived from the word chronos, being greek for time, is the *daemon* (or background monitoring process) in Linux that checks constantly for scheduled commands in each user's **`crontab`** file and then executes them. For our purposes it is the **`crontab`** files that will be edited by the user to specify **`cron`** jobs (i.e. scheduled tasks).

To add or edit a crontab entry by user, the relevant user simply needs log in and type **`crontab -e`** at the shell prompt (or, we could administer accounts via root, and switch to the account of the user in question, then type **`crontab -e`**).

Note that the default editor for crontab is **`vi`**, however if the user prefers another text editor this can be changed from the default by running the following command **`export VISUAL='nano',`** the text editor **`nano`** (based on **`pico`**, which is more or less equivalent) being much simpler for simple editing needs.

For the purposes of our automatic betting examples, we will use the crontab file to specify static scheduling tasks, or tasks which we know we can run at the same time every day. Let's look at an example and briefly cover the syntax used to specify scheduled tasks at the same time.

Example:
We want to schedule the program *`get_betfair_races.pl`* to fetch daily events and daily event summary details from Betfair - as discussed in *Example 5-2* in Chapter 5, *Betting Process 101: Daily Event Data*.

Our requirements are simply to run this every day before racing begins, and before any interaction with Betfair that may be required by any of our betting strategies.

Let's say then that the program will run at 8 am every morning, and that this is the first requirement for any betting strategy. In this case, our first entry in **`crontab`** will read:

```
00 08 * * * /home/aeb/get_betfair_races.pl;   #replace /home/aeb/  with your system path
```

Left to right, this states that the program should run at zero minutes (00) in hour 8 in the 24 hour clock (08), every day of the month (*), every month (*), every day of the week(*).

The syntax for crontab therefore has five fields for specifying time frequency, left to right, with permissible values shown in brackets:

- minutes (0-59),
- hours (0-23)
- day of month (1-31)
- month (1-12)
- day of week (0-6, e.g. Sunday = 0)

A wildcard * value has to be specified where the instruction is to mean every or "all".

This scheduling syntax is followed by the full path name of the command or executable file (in this case a Perl program) which is to be run. In the case of our example the program to go fetch Betfair events will be run at 8 am every morning (provided our system time is set correctly, as per the next section *Setting The Daily Time*), writing the events retrieved to a database, so we can reuse them later. Specifically, as we will discuss in *Dynamic Scheduling for Race Specific Programs,* we will go on to generate a file of times from the database that we can then use for scheduling other programs.

At

The **`at`** command can be used to schedule a command or executable file (e.g. Perl program script) to run once per command. As such, it is an appropriate command to use to schedule one-off dynamic scheduling tasks per race.

By default, only the root user can use the at command. However, the root user can use two files to allow regular users the command:

/etc/at.allow: If this file exists, only users listed in it can use the command.
/etc/at.deny: If this file exists, users listed in it cannot use the command.

The time for a task can be set using several formats. The simplest is a 24 hour clock in **hh:mm** format, followed by the date in **mm:dd:yyyy** format. If the date is not set, the command will be run whenever the time is next reached. We will be using the **`at`** command typically to schedule programs to be run within the current day, so simply using the time format.

We also use the **`–f`** option frequently to indicate that **`at`** should execute a file which contains other commands. The file must be executable, as must any commands, programs or scripts within the file.

The syntax is therefore of the form:

```
at –f filename time
```

Broadly speaking, we use **`at`** for dynamic scheduling and **`cron`** for static scheduling (where jobs do not need to be determined by daily race times). **`at`** is useful for scheduling around times that change since our programs can themselves write out

scheduled commands on the fly, by using the **`-f filename`** syntax as we will see in *Dynamic Scheduling for Race Specific Programs*.

Two other options in the filename that are useful for controlling **`at`** jobs are:

`atq` which lists the queue of at tasks for the current user, or all **`at`** jobs scheduled on the system if typed by the root user.

`atrm` which removes all jobs or tasks which have been scheduled by the current user, or all jobs if scheduled by the root user.

System Log Files

Since any scheduled task is usually executed by daemons run from root on behalf of named users, as in the case of **`cron`** and **`at`**, the root account will also send a message to the user's mailbox – location of which varies according to the user setup and Linux distribution (consult your documentation) - but usually somewhere like */var/username/mail* or *home/username/Maildir*. Such files record a log related to running the task in the form of a mail sent from the user as well as any error messages normally sent to the user terminal or standard output (`STDOUT`) relating to the scheduled task. The log file also captures any program output that would normally be sent to the user's terminal, or `STDOUT`.

Thus, any output from the Perl program that would normally go to `STDOUT` when running interactively is captured here instead when that Perl program is run as a scheduled task. In the case of our example to fetch daily races this output would be a list of event times and courses, since we print this to `STDOUT` for each line prior to writing it to the database. Capturing such output from the logfile (in general) and then being able to review it (programmatically if needed) is a useful debugging tool and enables us to check that programs have executed as anticipated. When running automated systems, such records are indispensable.

Setting the daily time

If everything about automation relies on timing, it is important to ensure that before we do anything else, our computer clock is telling the correct time.

Like most timekeeping devices, computers lose or gain time on a daily basis. Again like most watches, the amount of time they gain or lose can depend on the age of the timepiece.

For example, one of the home PCs I use, and which ran my first automated betting programs, is getting on a bit. Built from top of the range components in late 1998 with an early version of Linux installed, it now looks lame compared to what can be bought as a low end modern computer at the bottom of the range in any high street – although the components, despite being lower specification, may be more reliable.

Specifically, and getting back to the point in question, the old computer loses an average of 20 seconds a day if left to its own devices. This soon mounts up – so resetting the time, especially for any jobs which are scheduled in relation to real events, is therefore a necessity on a regular basis - best practice being to do this daily well before any time critical tasks begin.

A useful utility for the purpose of resetting the computer clock is **`ntpdate`**, which takes as a specification the name of the server which is telling the real time, whatever that is…

You should consult your system documentation for this, however, one way of maintaining this is to save the following command within an executable file:

```
Ntpdate -b time.apple.com
```

Where the executable file is called **`time.cron`**. Plenty of other public time servers can also be used to synchronise the computer clock, such as the pool of servers at *http://www.pool.ntp.org*. Take your pick.

The file containing the command should be set to execute daily, at any time of the user's choice, preferably before any scheduling which is determined by the day's timed events begins. We store this entry, since it will apply to (and be needed by) all user accounts, in the crontab for the root user.

Dynamic scheduling for event specific programs

We often need to automate the execution of programs for a betting strategy where the programs to be run (such as revising tissue prices based on non-runners before the off) are completely dependent upon the time of the events, and yet the event times themselves change daily. How to do this automatically?

Our approach to setting up any strategy which is dependent upon event times is to start with a list of events for the day that we are interested in betting. That list itself must be automatically generated in order for our process to be completely automated. As an optional step, we can filter the list for any further qualifying features before arriving at our final list of events. We then use the final list of events to generate a dynamic schedule.

Note that the examples provided later in this section refer to subsequent stages of the automated framework, so readers may wish to revisit this Chapter again after other programs within the framework have been covered in the rest of Part 2.

Creating the Events Schedule

We will assume for this example that we already captured relevant event details for the day from Betfair automatically, using the program *`get_betfair_races.pl`* which has been scheduled to run using the example crontab entry in the previous section, at 8 am

every day. We now need to repurpose the captured data as a list of times that we can manipulate within other programs to create an automated, dynamic schedule for our betting strategies.

For the time being, we will also assume that we are not interested in applying further filters to the list of races at this stage (e.g. we could if we wanted filter the list from Betfair based on the race description to either explicitly exclude or include courses or race types such as handicaps, sellers, maidens etc). Rather, that we have a strategy in place that will enable us to assess every race of the day against predetermined criteria and potentially to bet on every race, provided that certain conditions are met.

For illustration we will also use the specific list of events captured for 13th October 2007 and shown in *Figure 5-1* in Chapter 5.

Within our program, the next step is to capture this list of events by programmatically accessing the database, and then to order the list by time, showing the time of the race and the name of the course.

Our objective is to create a file containing this information that can be read by other programs through the day to determine which event they should be processing at which time. When racing begins, the same file will be reused by each set of scheduled programs, which will also rewrite the file having processed it, with one less event each time an event has passed. We add to the file the course name and the **marketId**, since our programs can later read in this information from the file and save extra processing steps. However, it is irrelevant as far as creating the dynamic schedule is concerned: all we are interested in is the sequence of times.

We have a number of choices in automatically generating the sequence of times described. In this case, however, we will issue a database query from within a Perl program and write the results to a file. The meat of the program is straightforward, as shown in *Example 7-1*.

Example 7-1: Generating a file of event details ordered by time (generate_events_schedule.pl)

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#


use DBI;
use strict;

#       Open the database handle
#substitute your database and user credentials
my $dbh = DBI->connect("DBI:mysql:autodb", "username") or die ("Error:  $DBI::errstr");

#       Issue the query, based on automatically using today's date:
#       Retrieve today's date using 'localtime' functions and convert to Betfair format
my $year = ((localtime)[5] + 1900);
my $month = ((localtime)[4] + 1);
my $day = (localtime)[3];
my $date = "$year-$month-$day";
```

```
open (EVENTS, '>/home/aeb/events_schedule');  #replace with your output path
my $sql = qq(SELECT DATE_FORMAT(time, '%H:%i'), course, marketId FROM races WHERE
date="$date" AND country = "GBR" ORDER BY time);
my $query = $dbh->prepare($sql);
$query->execute;

while ( my ($time, $course, $marketId) = $query->fetchrow_array) {

print EVENTS "$time, $course, $marketId\n";

}
```

Note the use of the `DATE_FORMAT` command in MySQL which enables us to display the time in HH:MM format, as opposed to the format in which the event time is stored in by the database (HH:MM:SS). HH:MM format is of course more appropriate to actual event times and also to scheduling using **`at`**, since seconds are superfluous.

The directory to which the file is written is the home directory for our default user */home/aeb*. There is no significance in this path, all the example programs and files are shown as written to the */home/aeb* directory. The file could of course be written to any directory, or even multiple directories, depending on what we want to use it for.

Below we use the **`nano`** text editor to display our file, so typing at the shell prompt

```
>nano /home/aeb/events_schedule      #replace with your system path to events_schedule
```

Will show us our events schedule for the day separated by the newline terminator as follows…

```
13:10, Ascot, 20633129
13:25, Chepstow, 20633172
13:45, Ascot, 20631295
14:00, Chepstow, 20633174
14:10, York, 20634370
14:15, Hexham, 20633191
14:20, Ascot, 20631297
14:25, Bangor, 20634384

#------rest of file omitted...
```

There are many other options as to how we could generate this file such as using our original program to write out each event, time and Betfair market ID to a file, sorted by race time, at the point of retrieving events rather than as a separate step.

Whatever way we go about achieving this, we will simply end up with an automatically generated file which lists one line per event. For the sequence of events alone, this could be generated from any other data source for daily events (such as electronically available racecards within a form database), provided it is afterwards sorted by time. We want above all to preserve the order in which the events occur in order to generate a schedule of tasks that consider each event in turn. For the time being, we'll add the generation of the ***`events_schedule`*** to our list of scheduled tasks in crontab, as the second entry to be run 10 minutes after retrieving the daily list of events.

```
10 08 * * * /home/aeb/generate_events_schedule.pl;  #replace /home/aeb/  with your system
                                                     #path to your program name
```

Generating dynamic tasks using the events_schedule file

In this section, we look at an example scenario for a betting strategy (described in the next subsection), to illustrate scheduling programs for execution in relation to changing daily event times. The individual programs are all drawn from the general framework and detailed further within Part 2. The programs are grouped together as the dynamic part of the betting strategy in order to execute as one. In the subsequent section, *Example Dynamic Scheduling Program*, we describe programs that schedule the dynamic part of the betting strategy on a daily automated basis, despite the changing times.

Example Scenario:

The example strategy is based on generic programs taken from each stage of the framework, which perform the following tasks: choosing events to bet in; assessing or selecting contenders to bet on within each event; capturing market information for those events; making a betting decision based on the assessment of chance and market information, and, finally, executing bets. We will not dwell on the merits of the strategy, rather concentrating on the overall process for automation.

Essentially, we want to schedule some programs to analyse and bet in each event, assuming that each program must play its part in the strategy according to each event's starting time.

Generically, our betting strategy has to do the following:

- Create an oddsline for every race in the UK based on applying a pre-existing model to daily racecard data and each horse's history, as discussed in Chapter 6. In the case of our illustrative example, we use data from the Racing Post website, but ideally this would be generated from a model based on a comprehensive database, or a third party oddsline based on the same – as illustrated in Part 3.
- The oddslines for every race in the UK are generated every morning before racing, the programs for doing so being scheduled by a **`cron`** job.
- Get market prices for each contender in every UK horserace 5 minutes before the race's starting time
- Immediately compare market prices with the oddsline tissue prices
- Decide whether or not to place a back bet. For now the criteria are not relevant, but let's say this is based upon determining whether or not an oddsline selection is "an overlay" or value bet.
- Place the bet

An example crontab file showing the order of entries to automate this betting strategy as described is listed in the last section of Appendix 1.

Scheduling every program in the strategy above (after the creation of the oddsline) to execute at the correct time before every event is the challenge that we will address in

this section. Subsequently, we will address the challenge of ensuring the scheduled programs choose the right events.

We assume we have a simple file containing our daily horseracing events, one event per line, as in the example ***events_schedule*** file created by the program ***generate_events_schedule.pl*** presented in *Example 7-1*. It is also assumed that any event that made it into this file is an event we want to analyse and potentially bet in.

In this case, "the correct time" for our betting strategy has been determined as 5 minutes before the race. The example programs to be executed (all of which will become more familiar in subsequent Chapters in Part 2) to implement the strategy as a whole are as follows:

- ***get_market_prices.pl*****:** ***get_market_prices.pl*** is the core Perl program shown in Chapter 8, *Capturing and Using Market Information*. It captures and writes out market prices for a given event to a file called ***inter_prices*** that can be evaluated by other programs afterwards (or the prices can be held in a data structure in memory when creating composite programs). This program is set to run in the strategy 5 minutes before every event.
  `get_market_prices.pl` must have a way of knowing which event it is getting prices for. The `events_schedule` file is therefore altered after each event is read in by the program (eliminating the event just gone) so that the next event to consider, the next time the program runs, is always the first in the file. We cover the steps required for this in more detail in the next section.
- ***bet_formation.pl*****:** This program is run immediately after `get_market_prices.pl`, reading in the file of prices for the appropriate event (from `inter_prices`) and evaluating it as a hash. It also reads in an oddsline for the event from a separate file such as our ***formatted_oddsline*** (the output from running *Example 6-1*), and compares each tissue price to that of the market price, determining whether or not the price for the contender is an overlay or underlay, according to the rules of the betting strategy. Clearly the oddsline file is a prerequisite to the strategy, and must be made available by a separate process that can be scheduled on a permanent basis (i.e. generating the oddsline can be done before racing begins). However, one of the reasons why the strategy is time critical, and therefore executed close to the off, is that `bet_formation.pl` will find any non-runners from the actual market compared to the oddsline (which is based on overnight declarations). The oddsline tissues presented by `formatted_oddsline` are then revised to account for any non-runners and made available within the program. `bet_formation.pl` outputs a file of bets for pick up by the next program in the sequence and is the type of core program we will see as an example in Chapter 9*, Automating Betting Decisions*.
- ***execute_bets.pl*****:** Our final program reads in the selections file created by `bet_formation.pl` and executes bets based according to those criteria. This program is discussed as an example in Chapter 10, *Betting Execution*.

Before we schedule these programs to be executed by the operating system, they are grouped to a single executable file called ***bet_oddsline_strategy***. There is no

particular reason to run 3 programs (as opposed to one) for the production strategy, although both approaches will work Grouping them in this way means that each part can be illustrated, tested, and the output analysed, whilst developing a strategy, and it is possible to run them sequentially to execute that strategy. Later we amalgamate them to remove redundant elements (as we discuss in Chapter 13). Here, the file listing the 3 programs will be executed as a command by the shell and thus needs to list the tasks we want to run, in the order that we want to run them. In this way the listing for ***bet_oddsline_strategy*** is as follows:

```
#Replace with your own system path

/home/aeb/get_market_prices.pl;
/home/aeb/bet_formation.pl;
/home/aeb/execute_bets.pl;

#Nb. One program in the sequence after get_market_prices needs to include code to move
#the events file events_schedule on to the next event, for the next time this set of
#programs is executed
```

In other words, all three programs will be executed consecutively when the file is executed. If this were just one program the principal would be the same – that is, executing the all the steps needed at one point in time. Now we know what steps we are trying to execute in one go, let's look at scheduling them.

Example Dynamic Scheduling Program

Given that the dynamic part of our strategy has been defined, as explained in the previous subsection (and per the programs in the executable file ***bet_oddsline_strategy***), let's look at one way of scheduling this part of the strategy dynamically, prior to the start of racing each day, so that it will execute 5 minutes before each event, based on each day's event schedule without manual intervention.

First, we will write a program solely for the purpose of executing the dynamic schedule. This time, in contrast to our ***events_schedule*** file which simply lists static event details, the program will translate the event details to a series of **at** commands, write these to an executable file and then have the operating system execute that file. This last step will commit the scheduled tasks to be executed by the operating system at the appointed time.

This schedule will be dynamic, meaning it will change every day, based on the changing times of events. However, the program which calculates the times for the programs to run and commits them to the operating system will be static, meaning it can be scheduled to be run at the same time every day, provided this is always before racing begins.

Let's call the program which does this *write_schedule.pl*. So, we need to add this program to our **crontab** file, running it daily at 8:15 am, for example, well before racing begins at any time of year:

```
15 08 * * * /home/aeb/write_schedule.pl;     #replace /home/aeb/  with your system
                                             #path to your program name
```

Bearing in mind that our strategy requires that all the programs in *bet_oddsline_strategy* are executed 5 minutes before each race, the *write_schedule.pl* program has to do the following specific tasks:

- Take in each event time and calculate the time that is 5 minutes before it
- Write the correct format for each **at** job out to a file ***execute_schedule*** (that can be read by the shell to execute the program before each event).
- For each **at** job we need to specify a time that is 5 minutes before our event time in the correct format (HH:MM), and specify the command we want to run at this time (in this case *bet_oddsline_strategy*).
- Run two Perl **system** calls to Linux shell commands from within *write_schedule.pl*:
    - The first command makes our newly created file *execute_schedule* executable by the shell.
    - The second command is simply to execute the file with all our scheduling information detailed as **at** jobs. At this point *bet_oddsline_strategy* will be written as a command to be run 5 minutes before each event as planned.

Prerequisites for *write_schedule.pl* are that the executable file which groups the programs to be executed must have been created – i.e. the file ***bet_oddsline_strategy***, containing the listing of the 3 programs described in the example scenario must have been created.

Example 7-2: Creating a dynamic schedule (write_schedule.pl)

```
#!/usr/bin/perl -w

#-----------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.                #
# The code from this example is provided under the terms of the Artistic License 2.0     #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code         #
#-----------------------------------------------------------------------------------------#

use strict;

open (IN, '/home/aeb/events_schedule');
open (OUT, '>/home/aeb/execute_schedule');

my @times = <IN>;

foreach my $event (@times) {

# get time for each event
my ($time, $course) = split(/,/, $event);
my ($hours, $minutes) = split(/:/, $time);

#  format time for output 5 minutes before start of race
my $ante_post = 5;         #time to set strategy in minutes before race

my $total_mins = ($hours*60)+$minutes - $ante_post;

my $schedule_hour = int($total_mins/60);
my $schedule_minute = sprintf("%02d", ($total_mins % 60) );        #gets remainder

my $schedule_time = "$schedule_hour:$schedule_minute";

#  format "at" statements to execute strategy program 5 minutes before start of each race

print OUT "at -f /home/aeb/bet_oddsline_strategy $schedule_time\;\n";
```

```
}

close IN;
close OUT;

system("chmod uo+rwx /home/aeb/execute_schedule");
system("/home/aeb/execute_schedule");
```

If this program is run manually from the command line as a test, a whole bunch of **at** commands fly past as the operating system schedules the batch command *bet_oddsline_strategy* to execute for each event time. We have a friendly warning that each command will be executed by **/bin/sh**, which tells us that a separate shell will be invoked to run the **at** jobs in the background come execution time, and also that there will be no output displayed when these commands run – instead the output will be directed to the system mailbox of the appropriate user running the jobs.

To check our dynamic schedule at any point, we can list the current **at** queue, by typing **atq.** The command **atq** lists all task details, one per line according to the job id.

The syntax for deleting a job is of the form **atrm** *job*. In the case of testing and generating what may be many instances of **at** jobs by running *write_schedule.pl*, therefore, it is useful to note that we can delete the entire **atq** – assuming all jobs are definitely unwanted, noting that this will delete all **at** jobs for the current user - with the following Perl one-liner (so called for Perl programs invoked from the shell, using the command **perl** with the **-e** argument).

```
atq | perl -e 'while (<STDIN>){ ($job, $rest) = split /\s+/; system("atrm $job"); } '
```

Although the programs within *bet_oddsline_strategy* will now execute 5 minutes before each race as planned, we have more work to do before *bet_oddsline_strategy* will work correctly whenever it is executed, within the context of the programs themselves. Specifically, we need to ensure that the generic programs within *bet_oddsline_strategy* will pick up the correct event details each time they are executed.

The next section explains one way to do that by reading in details of every event within the ***events_schedule*** file for each program in the strategy, and deleting any event from the file that has already been processed.

Picking the right event per scheduled task or program

Now we have a list of events that we wish to bet in, and a number of programs that are going to execute at the scheduled time before each event. However, each program, since it is general (i.e. written to take any event as an input rather than a named event), must still be able to locate the event corresponding to the time it has been scheduled for, then apply its part of the betting strategy to it. We need some generic code to do this, which will be run within the context of each program, and can again approach the problem in a number of ways.

In the case of this example, we will work from the event file ***events_schedule*** that we already have gone to the trouble of producing, relying on the fact that we are betting each event in sequence, with the earliest event always listed first.

One way to approach using this file would be to leave it untouched and create a counting mechanism to keep a tally of where we are in terms of events passed and "to go", every time one of the programs accesses *events_schedule*. However, doing this adds more complexity, such as storing a counter in an external file, and also removing the file acting as the counter before the end of the day in order to ensure it is ready to start at zero again.

For a simpler solution which also enables easy testing and monitoring, we will simply read in the list of events from the existing ***events_schedule*** file and treat it as a Perl array within each program. Then, when every program within the betting strategy is done reading event details for that scheduled time, we will write out the contents of the array again, this time as a file back to the hard disk with the same name, *events_schedule*, to be picked up at the next scheduled time - minus the event we have just processed.

Provided that the *events_schedule* file is only reduced by one event each time it is accessed, we can be sure that the correct event is always first in the queue. This will happen until there is one event left, corresponding to the final scheduling of the program. Thus, synchronisation is achieved between the scheduled programs and the events that are being bet.

To review some code that does this at the program level, let's take the retrieval of market prices. When we discuss this program in its core form in Chapter 8 we "hard code" a specific event within the script for the example, as follows:

```
#       For this example we pick one event and type it in, normally this is an
#       automatically created variable

$marketId = "20625238";                 #use your example event ID to test the program
```

Instead, to automatically schedule the capture of market prices for an event, we want to replace the code above in order to make it generic; in other words, so that the program can work out for itself which event it should be betting on whenever it is scheduled to run. The example code snippet below will enable our scheduled program to choose the first event from our file and use the **marketId** it picks up as the current one, so that no manual intervention is necessary. Whilst the purpose of the code is for appropriate inclusion within any of other program (where automatic capture of the most current event is required), it can also be tested "as is" prior to inclusion.

Example 7-3: Generic code to include in programs to capture current event details

```
#!/usr/bin/perl -w

#---------Start of program, prerequisite modules etc. omitted

use strict;
```

```
open (EVENTS, "/home/aeb/events_schedule");

my @events = (<EVENTS>);
close(EVENTS);

#the first event in the file (and therefore the array) always refers to the current event
#subsequently we will remove this event from the file, so that the next event will be
#first again, for the next time the program is executed.

my $event_details = shift(@events);
$event_details =~ s/^\s+|\s+//g;        # to get rid of dangerous invisible characters
my ($time, $course, $marketId) = split (/,/, $event_details);

print "$marketId\n";  #test we have read in the correct event
#Since $marketId now exists as a variable we can do what we like with it

#---------End of code to read in right event details from events schedule file
```

Now we have the **$marketId** available as a variable in the program we can use the rest of the code in ***get_market_prices.pl*** (or adapt it as required, depending on the frequency of prices required, as discussed in Chapter 8) in order to retrieve prices for this event automatically.

In our example strategy, once the *get_market_prices.pl* program has finished, our group of scheduled tasks in *bet_oddsline_strategy* continues by running the second program in the sequence, ***bet_formation.pl*** (*bet_formation.pl* is an example program described in more detail in Chapter 9, *Automating Betting Decisions*). The output of *bet_formation.pl* is either a number of selections which will be read by the next program in the sequence (***execute_bets.pl***) or nothing at all, in the case the program determines that there is no viable bet for this particular strategy.

As with *get_market_prices.pl*, in order for *bet_formation.pl* to work automatically, we require a given course, time and set of prices to make decisions on whether or not to place a bet. In order to schedule *bet_formation.pl* to take those decisions dynamically, we must again give the program the ability to find the event for which it has been scheduled. Again, we do this by reading in our file, ***events_schedule***.

As in the case of *get_market_prices.pl*, we require details for the current event (i.e. the event due to start in 5 minutes) from *events_schedule*. However, for ***bet_formation.pl***, our concern is with retrieving the variables for the course and time (as opposed to the Betfair market ID, which was the critical event information required by *get_market_prices.pl*) for the current event. This is so that the program retrieves the correct oddslines for the race from our ***formatted_oddsline*** file (produced in *Example 6-1*), which it does by matching the course and time from *events_schedule* to the course and time for all races in *formatted_oddsline*. Subsequently, we extract and compare tissue prices from the relevant oddsline contenders with actual prices from Betfair. However, *bet_formation.pl* does not need to connect with the Betfair API to do this, since the prices were already collected by *get_market_prices.pl* and made available locally within the file ***inter_prices***. Betfair information is required by subsequent programs to write out any recommended bets – namely the Betfair event id and the selection id – that *bet_formation.pl* may decide to make, but this information is also available from the ***inter_prices*** file we

are reading in. This is fine, we can pick and choose what we want from each line in ***events_schedule***.

However, there is more to do with ***events_schedule***, since, unlike the case of *get_market_prices.pl*, we not only want to read from it, but to rewrite it. Why? Since ***bet_formation.pl*** is the last program in the sequence that will need to read the current event within this betting strategy, the events file needs to move on for the benefit of the next scheduled execution of ***bet_oddsline_strategy*** and the next iteration of these programs, so that it is ready for the next event (or race) time in the sequence.

Note that ***execute_bets.pl*** (which is in fact the last program in the *bet_oddsline_strategy* sequence, based on the example program outlined in Chapter 10) will simply read from a recommended bets file written by ***bet_formation.pl*** called ***selections***, and place bets using the Betfair API. Since the *selections* file itself contains all the information needed to execute bets on the current event, there is no need for ***execute_bets.pl*** to interact with the *events_schedule* file to retrieve that data. The *selections* file will be overwritten by the next scheduled instance of ***bet_oddsline_strategy***, so ***execute_bets.pl*** will always receive data relating to the event it should be betting on from here, not from *events_schedule*.

As above, there is no need to preserve the current event information past the point at which ***bet_formation.pl*** has gathered it. However, ***bet_formation.pl*** does have a service to perform for all the other instances of ***bet_oddsline_strategy*** that are scheduled to run according to upcoming race times. That is, to delete the event whose details have just been read by the program, so that the first event in the list will be the event that will be considered by the next iteration of ***bet_oddsline_strategy***.

Example 7-4 shows the code for one way of doing that, i.e. saving the current event data for the program using it, but deleting it from the list and writing out the list again, minus the event just captured, so that the next scheduled considers the next event. Whilst the purpose of the code is for appropriate inclusion within any of other program (where automatic event capture and deletion is required), it can also be tested "as is" prior to inclusion.

The code shares the same starting point as the code snippet we just saw in *Example 7-3* except that this time we also open up a temporary file for writing out our new list of events, minus the one we are using, as in **open(TEMP '>**. The **shift** operation is still applied to the array created after we have read the ***events_schedule***, in order to capture the current event information held by the first value of the array. However, since shift also automatically reduces the array by its first element, the array is ready to be written back out to disk.

Finally, we write out the data in the array to the temporary file by setting up a quick **foreach** loop to print out the new array, followed by using the **rename** function to replace the current data in ***events_schedule*** with the new data that was held for a moment in the temporary file. The ***events_schedule*** file is now ready for the next scheduled instance of ***bet_oddsline_strategy***.

Example 7-4: Generic code to capture, use and then remove current event details for a scheduled program

```
#!/usr/bin/perl -w

#---------Normally include in betting strategy program, this an example of dealing with
#          events only
#---------Can be run "as is" to demonstrate the principal of polling through the
#          events_schedule file

use strict;

open (EVENTS, "/home/aeb/events_schedule");
open (TEMP, ">/home/aeb/temp_schedule");

my @events = (<EVENTS>);

my $event_details = shift(@events);
$event_details =~ s/^\s+|\s+//g;        # to get rid of dangerous invisible characters
my ($time, $course, $marketId) = split (/,/, $event_details);

#check we captured the right market, which will subsequently be deleted from the schedule
#can comment out next line if incorporating within betting programs
print "$marketId ($time, $course) is the current marketId, and will now be removed from
events_schedule\n";

foreach my $event (@events) {
print TEMP $event;
}

close(EVENTS);
close(TEMP);
rename("/home/aeb/temp_schedule", "/home/aeb/events_schedule");

#temp is the file with the correct data, but we want to rename it since it is expected by
#the next program, again with the name events schedule
#The variables - $time and $course - expected by bet_formation are now available within
#the context of the current program and the file events schedule has been rewritten for
#the next scheduled instance of bet_oddsline_strategy.
```

For a fleeting moment the ***events_schedule*** file is being used within the program and is also accessible at this time for alteration by other programs. Usually there is no concern here unless event times clash (which does happen on busy racedays such as bank holidays and some Saturdays), or programs accessing event times clash, so to be safe we can also lock the file (using **flock**) for all programs that attempt to access it.

Alternatively, we can split the events schedule into different directories per meeting (where each directory or meeting has its own *events_schedule* file) to avoid conflicting racetimes occurring, or wrap more elements within the control of single programs - as we do for the production runs of betting strategies in Part 3. Nonetheless, the principles of scheduling the combined programs according to event times, with a controlling file to monitor events that have been bet, remain the same.

An alternative scheduling case for simple system or third party picks

As a brief aside to our process for scheduling programs around each event, let's briefly consider an alternative for the case of simple betting strategies where scheduling needs may be more simple. For example, this may occur where there are few events to consider, the selections for each event are known before the race and, most importantly, if the need for time critical or market information to drive the betting strategy is minimal (e.g. the presence of non runners within the race will not change the selection strategy).

This can often be the case for simple betting system picks. Whilst it depends on the system, the general aim of selection systems as we defined them is to throw up individual selections based on fixed selection criteria rather than to compare multiple contenders in one race. Also, a rules based system identifying a specific combination of characteristics may produce few, if any, selections on a daily basis.

In such cases, the automation work to identify selections can be done in advance based upon racecard declarations, once the system rules are programmed into a database, or a program that interacts with the database. As such, the automation effort will be on programmatically applying system criteria to daily racing data (and of course creating the system in the first place), using an appropriately updated database. Once selections are generated from a data source, these can be combined with the static **marketId** and **selectionId** from Betfair within a database or a file so that selections are already prepared for betting. In other words, selections can be prepared well in advance of event times, with all information for betting execution in place.

Unless a system relies on some criteria that are time specific, such as the number of runners or the state of the going, it can be convenient to generate both the list of events for a specific betting system together with all the details necessary to bet the selections, making the Betfair calls for all static data early.

Once we are done with the systematic selection process from the database, together with all the Betfair details needed to bet those selections, the selections are ready to be bet. We can still create a list of events as discussed previously. The list of those events which are to be bet (even if the list is only one in length), can still be used for preparing to automatically schedule system picks, only the programs can take all their details from the file that is presented, as opposed to requiring separate input files such as an oddsline.

An alternative for a simple selection strategy is to execute all bets as soon as the file is ready, simply looping through the file, and executing bets (as described in *Betting Execution* in Chapter 10), thereby avoiding the need for dynamic scheduling altogether.

Such a strategy could place all bets for all system selections at once, no matter what the time, provided that the betting market is available. Whether or not it makes sense to place all bets for different races at once is up to the bettor's view of obtaining reasonable prices and the betting strategy itself. The danger with placing bets on the exchange well in advance of the market can be illiquid markets with large overrounds and thus poorer prices. On the other hand, the possibility now exists to be assured of

reasonable prices (not per selection but within the context of a reasonable overround) by executing bets at the Betfair Starting Price (BSP).

Betting at BSP has the disadvantage of execution price being unknown, although the extent to which that is a disadvantage depends on the strategy – e.g. it is not necessarily a disadvantage for systems where selections have no minimum price stipulation, although any strategy which does not consider price is itself suspect. For systems that do stipulate a minimum price, Betfair SP bets can be placed with a minimum price stipulation, as we cover later in Chapter 10. If that is the case for a simple betting strategy and the bettor does not think they lose an edge by betting in the exchange market, the betting process can be scheduled for all bets to be identified and automatically bet in one process, in seconds, every morning before racing begins.

Common Scheduling "gotchas"

For the avoidance of doubt, the recommendation is also to consult the operating system reference material if the reader is unfamiliar with automating tasks. However, there are some simple errors (though vexatious if a beginner) which can occur in getting started which are passed on here from the benefit of those engaged in early trial and error with scheduling.

1. Failing to specify either the full path name and/or the correct path name for the executable program in **`at`** or **`cron`** jobs, e.g. *`betting_strategy_oddsline`* instead of */home/username/betting_strategy_oddsline.*
2. Remembering the difference between executing files and executing shell commands from a permissions point of view. In the case of files, all the files must themselves be made executable. Use the **`chmod`** command to set appropriate permissions (type **`man chmod`** for all the arguments) and if in any doubt on a system where you are the only user, **`chmod ugoa+rwx`** ***`filename`*** – will make a file executable for reading, writing and execution by every user. Of course this will need more thought if there are other users on the system; we generally make the assumption that the reader is the sole user of the system through this book.
3. Use a step by step process for creating scheduled tasks in order to test and diagnose problems. Assuming we are running strategies on a daily basis, we could characterise the process for all tasks as prepare -> schedule -> test -> clean up:
   - **a.** Prepare: In particular, what are the dependent data, files or programs on which scheduling of the betting strategy relies?
   - **b.** Schedule: Suffice to say it is worth thinking out how each task will run in detail beforehand; as discussed in this chapter, sometimes it is necessary to schedule tasks with a degree of indirection, so that we need to schedule tasks that themselves schedule other tasks.
   - **c.** Test: When the programs executing the betting strategy are executed, what will happen within our strategy, step by step? It is worth breaking the programs and scheduling down into atomic steps and testing each part, so that the bettor is sure what will happen when the betting strategy is live.

d. Clean Up: Once a job is done we need to ensure there are no loose ends and that any contingent files are ready for the next betting cycle – as well as any end of day preparation which is necessary for the following day's cycle (e.g. ensuring all files are cleared down at the end of the day).

Extending scheduling concepts

Depending upon the imagination of the bettor and the complexity of the strategies and tasks that they want to implement, we can think up infinitely more complex cases than those shown so far. However, the chances are that the basic methods and tools already described, together with other facilities at our disposal, can more than keep pace with the complexity or number of betting strategies required. Let's therefore consider some of the ways of scaling and extending the concepts to keep pace with more complex scheduling requirements.

So far we considered one strategy applied to one set of daily events. The strategy had three steps, each step consisting of a short program, run consecutively. At this point, the betting strategy for that event was complete, and the schedule could move on to the next event.

What if our strategy had far more than 3 steps, such as a trading strategy, where we might want to revisit the same market again and again, and deal with partially matched bets? Our simpler strategy was executed in milliseconds for a specific event time and then done, with the `events_schedule` modified ready for set of programs to deal with the next event. What if we wanted to run a strategy for a market that itself extended over the time of other events, for example starting much earlier than 5 minutes before a race – how could we ensure it was automated and did not interfere with other strategies that had to start alongside it? What if, instead of running one strategy, we want to apply different strategies to different events on the same day, or even different strategies (e.g. a betting system, dutching an oddsline, price arbitrage etc) to the same event? How could we go about implementing all these in order to meet the objectives of complete automation?

Scaling for multiple strategies by using multiple directories and user accounts

Without doing anything much different outside the current framework and scheduling facilities already described, we can create many more possibilities to cope with increasing the number of automated betting strategies, simply by paying attention to careful organisation of the filesystem.

In the examples shown so far, we have run all events for a strategy from one directory, using as our parameters for automation the ***`events_schedule`*** file, containing all events for the day, so that the timing of **`at`** jobs is written around the events schedule for the entire strategy. As the number of different strategies and events grows, combined with potentially more complex strategies which are visiting the market more times, it will become more of a struggle to use a single file which is manipulated by each strategy to

indicate which event is next. There is the potential of file access conflicts, moreover of pushing forward to an incorrect event, or reading an event which has been incorrectly appointed. It will be a challenge for each strategy to keep track of which events have gone or are next if many programs are attempting to take their cue from them. However, there is nothing to stop us automating strategies and/ or events within discrete directories.

If common files are used for common strategies, such as our ***events_schedule*** example file, we can replicate such common files across directories. A static scheduling program can be set up in advance to copy files or populate multiple directories before racing begins, along with any other files needed for betting. The strategies themselves can be set to run by static scheduling programs within the directories within which they are contained, as we have already seen for one strategy.

The wisdom of other approaches really depends on the strategy, but we can accommodate any amount of complexity by setting up multiple directories, for example to cover one meeting each day – say ***meeting1*** through to ***meeting8***, and generate a version of the ***events_schedule*** file that creates a time ordered list of races on a per meeting basis, copied to each directory. In the case of a UK racing strategy there are rarely more than 8 meetings in a day. This gives us 30 minutes between events in each directory, without overlap, so we can read from the same events schedule and schedule many activities around each specific race.

An *end of day process* could pick up any relevant files from the directories, clean the directories up and prepare them for the next day's betting, or delete them altogether. Since we can create directories within directories *ad infinitum*, the issue with this strategy is less the operating system's capacity to keep up with the bettor's plans but the bettor's own capacity to organise appropriately. By and large, simple is best, not least due to the consideration of maintenance.

What about if we want to run several different strategies off the same event? As well as the organisation of directories, we can create different users per strategy. Since Linux is a true multi user system, everything is set up to treat a user as the owner of a different strategy, even if the bettor is the sole user of the actual system.

This can be a convenient way to schedule different strategies, either modifying entire directories of existing programs and files or creating distinct strategies, or trading activities, from scratch. We should bear in mind also that any scheduled process will send the output of its jobs automatically to each user's mail directory, meaning troubleshooting or log analysis is nicely separated by user account, which as we have said could represent different strategies. The whole organisation can be monitored by the **root** or superuser.

We will not go further at this point on the theme of creating directories and user accounts for scheduling purposes, since the wisdom of doing so is down to the particular set of strategies in question. Suffice to say the tools are at our disposal should we need them.

Handling events at the program level

As any betting strategy becomes more complex there is a decision point regarding what tasks are appropriate to handle through the scheduling facilities of the operating system and what tasks should be handled within the context of individual programs. The scheduling of any activity can be characterised by the need for the automated strategy to respond to time driven events, but for the most part there is nothing to stop these being handled by the programming language interacting with the operating system, rather than running separately to the program.

Although the framework examples generally demonstrate the smallest components within each stage of the betting strategy, programs inevitably become longer and more complex as betting strategies evolve, combining elements of the framework in one. When this happens, it can be simpler to handle all the data retrieval for a single event within the context of a single program, along with other time specific information and file inputs, rather than relying on separate programs to collect data. As ever, it depends on the strategy, but single programs can reduce the number of external files generated and the number of times we need to rely on passing on information about what event the program should be dealing with. Putting all event driven information into one program simplifies our scheduling task to kicking off the program at the appropriate time, rather than kicking off the program and several others, then monitoring which events each is acting upon.

Provided the program receives the required information once, it can act in relation to that event thereafter, and we can still maintain the stepping mechanism whereby the current event information is read once and the file is shifted to the next event. The fact that the program itself will become an ongoing process for a number of minutes in order to execute on the strategy is neither here nor there as far as the scheduling process is concerned.

As discussed for the examples above, we can initiate any number of activities within the context of a single program, revisiting the market several times, making multiple bets and multiple revisions. This can be achieved by keeping a program open to make many calls to the same event, and to program it to behave according to our strategy dependent upon that information. This will involve more careful programming, of course, for example to ensure that the session with Betfair is maintained correctly (using the correct returned session tokens from other calls to the API), and if the program can be idle for some time, using the **KeepAlive** service call to maintain the session as well as creating checks and failsafes in the case of failed calls.

From a scheduling point of view, several mechanisms are at the bettor's disposal to control how time is handled within the program, to the degree where we can create programs that behave as the bettor would manually.

Using sleep

Both the operating system and Perl put at our disposal respectively the basic command or function **`sleep`**, which is proceeded by a number, representing seconds of elapsed

time. Essentially **sleep** tells the program or shell how many seconds it must remain inactive before continuing with the next command or statement.

We have already used sleep within a Perl program for the ***get_betfair_races.pl*** program, where, in order to use the Free Access API for the example, it was necessary to wait for 12 seconds before calling another market. The **sleep** function was able to buy that 12 seconds, before repeating the request for another market. Likewise when used from the shell, **sleep** is often useful to wait awhile before executing the next command in a sequence of commands; those commands could equally well be programs.

Timekeeping within a program

For many of the programs in the framework, the examples are framed in small units to illustrate specific points, and are by and large executed and exited within milliseconds. However, keeping a program alive for a long period of time, executing many tasks and maintaining its own timekeeping can be a more common solution as strategies become more complex and there are more of them.

As we use the scheduling facilities less and rely less on scheduling multiple programs, so we rely on our one program to do more of the heavy lifting with regard to timekeeping - in other words, dictating what the betting strategy should be doing and when it should be doing it.

Of course, the program itself must still be initiated at the correct time, which is what we will still use **cron** to achieve, or an **at** job. The scheduling will be combined with a mechanism within the program to specify the course, event and time information which the rest of the program will use - such as the manipulation of our ***events_schedule*** file specified earlier.

Subsequently, we need to keep track of time with a view to the strategy, depending on what the strategy is. Let's throw a few possibilities together to make this more concrete - say we want to run a program to begin 10 minutes before the start of a race which will do any combination of the following, and to some level of repetition: track prices for a certain amount of time, make a predictive analysis, place bets, analyse which bets have been matched, track prices, monitor which bets have been matched or unmatched from the original set, make a decision on which bets to cancel or modify and so on, right up until the offtime. If the offtime is T and 10 minutes prior to the race is represented as T-10, then let's say that the strategy translates to knowing in advance that we want to track prices at T-10, make bets at T-9, track prices again at T-8, reconcile unmatched bets at T-7 and so on.

In order to know what T-N is equal to and set off tasks relating to it, we have to keep track of time throughout the program's execution. An example of tracking time in relation to an event (within a program) and corresponding walkthrough, together with executing tasks based upon that is shown in Chapter 8, *Example 8-4*.

Much can be achieved for any betting strategy at the program level using the facilities discussed, and for more esoteric tasks there are plenty more modules available within a Perl context, for timekeeping and event handling, available from *www.cpan.org*.

Summary:

We've looked at the basic tools available - scheduling using **`cron`** and **`at`**, scheduling for individual users, and basing scheduling programs on changing daily event information.

We've looked at ways of combining these tools, as per a general example to bet oddslines, based on the generic elements of the whole framework, to give us an effective automatic strategy. We've looked at ways we might start to scale further, using different directories and user accounts, as well as creating more complex betting programs.

It is impractical to cover every permutation which will be created by every betting strategy and therefore every way we might wish to schedule that betting strategy, but we can see through the raw material that we can start to contemplate any specific betting strategy which can be quantified - and moreover be confident in the potential to automate it.

Subsequent stages of the framework can be picked out and put together for any particular strategy. However, all will require an umbrella for automatic operation such as covered by the basics of scheduling and automation covered within this Chapter.

Chapter 8: Capturing and Using Market Information

Reviewing market prices is generally a prerequisite before proceeding with any betting strategy. In general, this stage in the betting process occurs after choosing an event to consider for betting and assessing the contenders within that event, or choosing a contender which conforms to some selection criteria. For an oddsline betting strategy, comparing market prices with the oddsline tissue is the foundation of the strategy. Even when betting a system religiously, a minimum price is usually required.

However, market information is wider than price alone. Although the most common form of market information in the exchange world is the current back and lay price for all given contenders in the market, there is much intelligence that can be derived from volume information and by going deeper into the market than currently available prices and volumes. Below, we recap on some of the most common uses for market information, before going into examples to illustrate this stage of the framework.

Whenever market data is captured, the automatic bettor has an opportunity to validate the health of the market itself in order to decide the timing of a betting decision. Typically we might calculate the overround for back prices or underround for lay prices, or the volumes or spreads between prices that is available to inform this view. For example, whatever the view of the bettor about the price of an individual contender, it is generally true that a market which is close to 100% will offer the backer much better opportunities than one which is not. An overround or underround calculation is therefore a useful piece of information to pass on to the betting decision making process, and can be derived at the market information stage.

It is worth noting that capturing and using market information can be a stage in the framework that is repeated again and again for the same event. This is self-evident for a trading strategy, but as above can also be the case in visiting the market multiple times to secure best price.

Looking at current back and lay prices, the volume available, and the “price ladder” for each contender (or the total amount matched, and the available amount to back and lay at every price on the “ladder” since the market began) can tell us much about trends and opportunities in the current market and therefore what strategies are best suited to playing it at any given point in time.

Similarly, interpreting the 3 levels of back and lay price information which are visible from the user interface (and via the API, using the **GetMarketPricesCompressed** service call) is a useful “quick take” on current trends, and can help us to determine market momentum or relative strength for each contender. Calculating average prices across all volumes is impossible to do on a timely basis from a manual perspective, but trivial if done with computer assistance, and easy to automate. Visibility to market trends is clearly critical for trading but also can help determine if we are to put in an offer for a better price than the current back or lay price available when betting on outcomes.

Every runner’s detailed trading history, together with all amounts to back and lay at every price range also constitutes market information. As with the 3 levels of immediate price information, interpreting this information quickly and consistently is tricky to

impossible from a manual perspective. For the detailed price history, it is also impossible to view for all runners manually, so using the API is a big advantage, depending upon how any analysis is programmed. From an API perspective, the required data is available via the **GetAvailableMarketDepth** and **GetTradedVolume** services that are reviewed later in the chapter. The newer function **GetCompleteMarketPricesCompressed**, to give all levels of liquidity for all runners, can also be used in this regard.

Outside the context of executing a particular betting strategy on an event, capturing and storing market information can also provide us with a useful resource to research, develop and test future strategies. In this case we could be looking to build up a comprehensive picture of each market at various time points, or simply to capture one or two sets of prices and volumes close to the off, depending on the objectives of the research.

For example, we might backtest a betting strategy using Betfair prices as opposed to SP, or we might be looking at different returns for betting strategies based on market timing. Such research culminates in the case of looking at market information alone for developing and testing trading strategies. Although betting on outcomes is our primary focus, the overlap between betting and trading strategies is nowhere clearer in the framework than when looking at capturing and using market information (we briefly discuss adapting the framework for trading in the last section of Chapter 9, *Automating Betting Decisions*).

Last but not least, obtaining market information also enables us to test any betting strategy prior to risking any money by creating "what if" scenarios that can subsequently be compared to a results database.

In the next section, we will look at example code for capturing prices and then consider what further value we can add to the information captured, plus how we can make the information available for further usage. Subsequent sections deal with running the code to capture market prices in practice; extending the scope of this function to look at weighted average prices, weight of money indicators and calculating overrounds, and finally, working with the complete price and trading history for all runners in a market.

Example code for capturing current market prices

The most fundamental information on the exchange is the current trading price and volume available to back and lay. In terms of API functions, this and other market information is accessible through one service call that we have already come across, **GetMarketPricesCompressed**. We reviewed this function in some detail in Chapter 4, in the section Review of the GetMarketPricesCompressed example function.

The arguments for using this function require a relevant **marketId** to be supplied. To automate, we pick from one of the relevant daily events, a process which we covered for retrieving daily markets in Chapter 5.

For convenience, we also show calling **GetMarkets** in conjunction with this function, since **GetMarketPricesCompressed** retrieves only the Betfair identification numbers of

selections and **GetMarkets** lets us retrieve their names also. Since we would like to present prices along with the real names of each contender, we use this second call.

The first example program uses these services to get prices for one market at one point in time. It is therefore a single, if key, component that needs to be expanded to be part of a practical strategy. Typically, it would be made generic to work for multiple timepoints and multiple markets, as well as automatically being supplied with the required market ID, without the need to explicitly state the market ID in the script.

We cover the elements relating to multiple markets and multiple timepoints in the next section, *Extending the Program*. Regarding automation of the program - both to automate execution of the program at the required time and to choose the correct market details automatically - we refer back to Chapter 7, *Scheduling – The Key To Automation*. *Example 7-3* in that chapter covers how to capture events automatically, so that the explicit **`$marketId`** in the script can be replaced with an automatically chosen one.

Back to our current example: We retrieve a set of market prices for a given market within a few lines code, and need to deal with the output, which can be the greater part of the script, depending on what information is to be stored and how granular it is. For example, if we wish to retrieve data for all levels of the market that are visible within the website interface, that increases our storage requirements (in terms of setting up the data structures required) by three times what would be needed be if using current back and lay, price and volume information only, if storing the whole "market ladder", significantly more.

In this example we will look at the most fundamental case, for the currently available market, and write these values to a simple database table, as well as "dumping" a set of values to a file as a perl data structure that can be read back in to another perl script. This is an alternative temporary data structure to using a dbm file which we looked at when using the events data (and one which will be useful if users are trying to set up on Windows, since the dbm file is a UNIX construct).

Setting up the database table to store prices is a prerequisite to running the script, although an alternative for data which is not intended to persist for more than one day is simply store prices in a flat file.

Example 8-1 creates a simple example table to store current market prices. This table is used in conjunction with *Example 8-2*, and other subsequent examples, for retrieving and storing currently available prices.

Example 8-1: Create price table within the autodb database (create_price_table.sql)

```
#CREATE price table for currently available Betfair prices within AUTODB database

DROP TABLE IF EXISTS price;
#@  CREATE TABLE
CREATE TABLE price
(
        marketId         INT(11) NOT NULL,
        runnerId        INT(11) NOT NULL,
        runner          VARCHAR(25) NOT NULL,
```

```
        date            date NOT NULL DEFAULT '0000-00-00',
        timestamp       time NOT NULL,
        back            DECIMAL(6,2),
        backvol         DECIMAL(10,0),
        lay             DECIMAL(6,2),
        layvol          DECIMAL(10,0),
        PRIMARY KEY (marketId, runnerId, timestamp)
);
#@ _CREATE_TABLE_
```

Assuming you do not have to specify a password to access your database (in which case that would be added under the **–p** flag), the command is:

```
> mysql –u username autodb < create_price_table.sql
```

Note that one of the fields used as the primary key for the table is the Betfair **`marketId`**. To obtain full data from queries on this table, we assume that we also have the **`races`** database table set up (as described in *Example 5-1*) in order to **`join`** the tables by that key and therefore display information on the course and time which any market prices relate to.

Example 8-2: Retrieve and store Level 1 market prices and volumes (get_market_prices.pl)

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee 2004-2008.  All rights reserved.                   #
# The code from this example is provided under the terms of the Artistic License 2.0        #
# Available for free download with full licence terms at http://www.betwise.co.uk           #
#-------------------------------------------------------------------------------------------#


#       Objective of script:
#       Fetch market prices for a given Betfair market ID, calling the
#        get_mArket_prices_compressed function
#       To test this script the event must be explicitly named, for production scripts
#        this is captured automatically using code from Example 7-3
#       Finally, save the market information to a database, and to a hash of arrays which
#       is dumped to a file for later use


#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";           #Free Access API access code

#       other program variables not declared in line
my $back_price;
my $lay_price;
```

```
my $lay_vol;
my $back_vol;
my $marketId;
my $timestamp;
my $date;
my $discard;
my %prices;
my %inter_prices;

#       open our database handle for a permanent record of the prices

my $dbh = DBI->connect("DBI:mysql:autodb", "username") or die ("Error:  $DBI::errstr");
#substitute your database and user credentials

#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }

#       For this example we pick one event and type it in to demonstrate price capture
#       Eg. Run Example 5-2 and choose an ID, then hard code it as below

$marketId = "20625238";        #your example market ID for testing price retrieval

#       To automate capture of the relevant event, see Chapter 7, and Example 7-3, or 7-4
#       Eg. if wishing to automate capture of the next event from an events_schedule file
#       Uncomment code below, within context of other programs in the framework

#       Code snippet to include from Example 7-3 for automating event capture
#open (EVENTS, "/home/aeb/events_schedule");   #use your path to your schedule of events
#my @events = (<EVENTS>);
#close(EVENTS);
#my $event_details = shift(@events);
#$event_details =~ s/^\s+|\s+//g;        # to get rid of dangerous invisible characters
#my ($time, $course, $marketId) = split (/,/, $event_details);
#       end of snippet for automating event capture

#       Get market information for this event, and save the array for runner names
my @market_array = get_markets($token, $marketId);       #runner_name is key, runner ID
is value
my %static_runner_data = %{$market_array[0]};
#runner_name is key, runner ID is value

#      Get current market prices
#       runner ID is key, runner prices are a value of hashes

my %prices_hash = get_market_prices_compressed($token, $marketId);
$timestamp = $prices_hash{timeStamp};
($date, $timestamp) = split (/T/, $timestamp);
($timestamp, $discard) = split (/Z/, $timestamp);
print $timestamp;
# N.B.:  hashes for static data and dynamic prices are now in memory, all market
# information now retrieved from Betfair

my @names = keys(%static_runner_data);
foreach my $runner (@names) {

my $runnerId = $static_runner_data{$runner};

$back_price = $prices_hash{prices}->{$runnerId}->{back}->{1}->{price};
$back_vol = $prices_hash{prices}->{$runnerId}->{back}->{1}->{amountAvailable};
```

```
$lay_price = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{price};
$lay_vol = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{amountAvailable};

#       test all captured variables are correct
print "$marketId, $runnerId, $runner, $date, $timestamp, $back_price, $back_vol,
$lay_price, $lay_vol\n";


#       Save the information now, first for a persistent record in our database
my $sql = qq(INSERT INTO price VALUES
('$marketId', '$runnerId', '$runner', '$date', '$timestamp', '$back_price', '$back_vol',
'$lay_price', '$lay_vol') );
my $query = $dbh->prepare($sql);
$query->execute;

#       also create a hash of arrays to represent our runner prices that we subsequently
#       dump to a file:
$inter_prices{$runner} = [$back_price, $lay_price, $lay_vol, $back_vol, $runnerId,
$marketId];

}

#       Dump out prices to a file which can be read into an example decision making
#       program later, as an alternative to database access

use Data::Dumper;
$Data::Dumper::Purity=1;
open (FILE, '>/home/aeb/inter_prices');
print FILE Data::Dumper->Dump([\%inter_prices], ['*inter_prices']);
close FILE;
```

Running the program

The example captures one set of current prices for a specific event (using the Betfair market identification number) and writes the output to a data structure which is held in memory for the duration of the program and is stored to a persistent format such as a database for reuse when the program has exited.

Retrieving prices for a single market using **GetMarketPricesCompressed** in conjunction with **GetMarkets** is the core routine for all the price programs, capturing names with prices, then extracting them to a database. This core routine is more useful from a practical point of view when the program is scheduled to run at the desired time and automatically captures prices for relevant events at that time (without having to explicitly state the market identification number to use). We discuss both these aspects of automation for the appropriate events in Chapter 7, since it applies to many other functions, showing the appropriate code to add in *Examples 7-3*, through *7-4*. In *Example 8.2*, the appropriate code is included in the correct position but shown commented out, so that the program can first be tested to show it works correctly for an explicitly named market, before being allowed to run automatically as a scheduled task.

Once we have called the `get_market_prices_compressed` function, we have the option of using all the levels of information returned and calculating derived variables, such as, for example, the average price and volume for back and lay prices. Typically, we might use changing information on average price to indicate momentum on one side of the market or the other, in conjunction with currently available back or lay prices. This data could feed through to our betting decision making process for further consideration. We consider calculating average price for all back and lay amounts to

the 3 levels of price that can be retrieved by `get_market_prices_compressed` in the upcoming section below, *Extending the program.*

Likewise, in terms of derived variables, we can use the market overround in our betting decision making process, and have all the information available to calculate it at this point. We also consider calculating the overround later in this Chapter at the point where prices are captured (it could also be done from the database). Once we have data relating to the overround, it can inform later stages of the framework.

Let's look at the output from this example (which is of course the same for the appropriate timestamp whether the program is run interactively or on an automated basis) by querying the MySQL table used to store the data.

Using the interactive command line in MySQL, we retrieve current price and volume information for each runner in the race, ordered by the back price.

```
mysql> SELECT runner, timestamp, back, backvol, lay, layvol FROM price WHERE
marketId="20634376" ORDER BY back;
```

runner	timestamp	back	backvol	lay	layvol
Dabbers Ridge	15:37:53	6.40	579	6.60	119
Fonthill Road	15:37:53	7.40	41	7.60	169
Commando Scott	15:37:53	12.50	115	13.50	59
Tawaassol	15:37:53	13.50	36	14.00	43
Dhaular Dhar	15:37:53	15.00	16	15.50	22
Rising Shadow	15:37:53	16.50	108	17.00	20
Conquest	15:37:53	17.00	156	17.50	33
Philharmonic	15:37:53	18.00	153	19.00	69
Zomerlust	15:37:53	19.00	52	19.50	6
River Falcon	15:37:53	22.00	2	23.00	40
Hoh Hoh Hoh	15:37:53	26.00	18	29.00	2
Viking Spirit	15:37:53	26.00	60	27.00	13
Turnkey	15:37:53	27.00	19	29.00	10
Obe Brave	15:37:53	28.00	3	32.00	42
Somnus	15:37:53	36.00	10	38.00	27
Tournedos	15:37:53	42.00	12	48.00	5
King Orchisios	15:37:53	50.00	474	55.00	4
Golden Dixie	15:37:53	65.00	13	70.00	40
Chicken Soup	15:37:53	75.00	8	80.00	3
Invincible Force	15:37:53	130.00	10	140.00	10

Figure 8-1: Market data ordered by back price for a single timestamp

Ordering runners by their back price is the same way that horses are ranked on the Betfair interface, with those trading at the lowest prices shown first. Ordering by the back price gives us a useful view on the market, as well as enabling us to automatically apply criteria where betting strategies require "knowledge" of where a runner stands in the betting, which is often the case for systems or trading strategies devised around the favourite in the market:

Extending the program

The program in *Example 8-2* can be scheduled to run automatically as it stands but every time prices for a new event are needed we would have to enter the new event manually. In practice we will always specify the events (for which prices are to be captured) automatically. How we do so depends upon our betting strategy and therefore what events we are interested in. Our scheduling programs therefore need to "know" when the events in the betting strategy are happening, without us specifying them on a daily basis, and what tasks (such as capturing prices at different time intervals) must be executed in relation to the event. This we explored in detail in the previous Chapter *Scheduling – The Key to Automation*.

Leaving aside the actual scheduling of the program in relation to the event, there are many options at the program level for extending the core routine to capture prices for multiple events in one program, and multiple price points for those events.

Market prices for multiple events

In this section we discuss the differences that enable us to capture market prices for multiple events, as opposed to a single event. The principle here is to capture a list of events within the program and then iterate through them one by one, capturing the required price data along the way. To illustrate automatically passing event details to our program, we could use our existing database handle to connect to the race events database table **`races`** that we created in *Betting Process 101: Daily Events Data,* then issue a query to retrieve all events.

Note here that we are using the Perl DBI module again to retrieve data via programmatic queries (that can thus be automated), as opposed to via the MySQL command line.

Creating a list of events and selecting from that or passing in each one would clearly mean replacing the "hard coded" event (a.k.a. market) ID from our script in *Example 8-2*, as below:

```
#       For this example we pick one event and type it in, normally this is an
#      automatically created variable

$marketId = "20625238";           #your example event ID
```

Instead, we can insert code to amend *Example 8-2* in order to capture market prices for all the day's events by substituting in the code in *Example 8-3* at this point.

Example 8-3: Capturing prices for multiple events

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee 2004-2008.  All rights reserved.              #
# The code from this example is provided under the terms of the Artistic License 2.0   #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code      #
#-------------------------------------------------------------------------------------#

#   Additional code for Example 8-2 to fetch prices on multiple events
#   Full example incorporating this code available at www.betwise.co.uk

#       Adapt Example 8-2 to automatically retrieve all prices for daily events
#       Delete hard coded marketId and instead query database for all events, as below

#       Get today's date

my ($day, $month, $year) = (localtime) [3,4,5];
$year+=1900; $month+=1;
$date = "$year-$month-$day";
#$date = "2007-10-06";

#       Form and execute query to retrieve Betfair event ids
my $daily_bf_events = qq(SELECT marketId FROM races WHERE date="$date");
my $IDs = $dbh->prepare($daily_bf_events);
$IDs->execute;


#       Pass each event ID to the main body of our market price capture program
while (my ($marketId) = $IDs->fetchrow_array) {

print "$marketId\n"; #check we are successfully retrieving the correct events

sleep 12;       #if using the Free Access API, to account for service throttling

#       continue where we left off before in Example 8-2
#       Get market information for this event, and save the array for runner names
};       #Remember to accommodate the new while loop outside body of Example 8-2
```

Note we have inserted the Perl code to get today's date, so that the date variable can be passed automatically to our database query.

Next, we set up a **`while`** loop to process each element of the array (each element is of course a separate Betfair market) and pass that to the main body of our example function, to fetch prices for each market. Finally, if adapting the code in *Example 8-2* to add this functionality, we need to remember to close the new loop `};` at the point in the original code before writing out (dumping) to the file the **`%inter_prices`** data structure.

Now that we are retrieving prices for all races, the emphasis is on storing to a database rather than writing out a file of the prices, so the *`inter_prices`* file can also be dropped. This file is simply a database alternative in the event of dealing with a light amount of data, and passing it conveniently to the next stage of the betting framework. We will, however, come back to using this data structure for consistency in running the example oddsline strategy discussed for ***`bet_oddsline_strategy`*** in Chapter 7 and run live in Part 3*, Automated Strategies in Practice.*

The modified program now enables us to retrieve all prices for all races that exist in the events table, in other words for every event on the day, depending upon the events criteria that we specified. We write out all runners and price information, together with a

timestamp, to our database table **price**. The organisation of the database tables could no doubt be improved and more error checking put in place, but in principal it works. Note that the **GetMarkets** call is subject to service throttling for the Free Access API of 5 calls per minute. So there is the addition of the statement

```
sleep 12;
```

..before each function call to the **GetMarkets** and **GetMarketPricesCompressed** services begins, within the context of a loop, so that we are able to use the Free Access API. This makes our loop wait for 12 seconds before passing in another market ID. If this is not done, the calls will simply fail due to throttling of the service. Alternatively, subscribers to read only services (since **GetMarkets** is a read only service), can delete this line, and the script will whiz through all markets in seconds or less, picking up all prices.

Running this program at any time before any event begins will consistently update our database with market prices for each event - although is somewhat inefficient if using the Free Access API due to service throttling.

Thereafter, events will disappear with increasing frequency until there is no data left to return. In this sense, the program is a scattergun approach rather than a specific one, attempting to grab everything at one point in time without much sense of purpose, returning void results for those markets that have lapsed, in-running prices if markets have turned in running and prices only for the timestamp at the point where the program is executed.

Without regular scheduling, our database would therefore give a very sketchy picture of what is happening in each market. Therefore, for a useful implementation, the program needs to be scheduled with a predictable frequency, and attention paid to markets passed. For example, we might specify the current time for the database query to events, and loop only through those that exceed the current time. Alternatively, we can manipulate the array of events within the context of the program as required – e.g. check the event status before collecting prices, otherwise move on).

Fetching prices at regular intervals for an event

In this subsection we highlight the adaptations required to our core price capture routine (shown in *Example 8-2*) in order to automate the capture of prices at regular, specified time intervals, as opposed to one specific time only, for any given event. As with the previous example for retrieving multiple events, the **while** construct is the looping mechanism used to repeat our core price fetching routine – on this occasion, repeatedly getting prices for the same event but at regular time intervals. As per the adaptation discussed in the previous section, any application of this program will seek to automate the event that is chosen by the program rather than hard coding a particular **marketId**. However, testing for different time intervals with one **marketId** is a useful exercise to determine that the program behaves as required prior to automating execution of the program and automating capture of market IDs.

On the basis that the core program to fetch each set of prices uses the code in *Example 8-2*, we can look at the principals involved to repeatedly call prices at given intervals up to a certain event time. We start with the code listing below and continue with a full explanation thereafter.

Example 8-4: Fetching prices at regular intervals up to a given event time

```
#!/usr/bin/perl -w

#---------Adaptation for Example 8-2 to fetch prices more than once on the same event
#---------See code walkthrough in book
#---------Requires specification of market Id and race time, usually read from dbase
#---------See Example 8-3 for an example reading in database variable instead

use strict;

my $race_time = "14:10";        #$race_time variable should be supplied automatically,
                                #either from a database or flat file
                                #The hard-coded variable 14:10 for $race_time is supplied
                                #only for explaining the example code in the book
                                #To test code snippet, run "as is", specifying your own
                                #race time close to the current time

my ($race_hour, $race_minute) = split (/:/, $race_time);

#convert the race time from HH:MM to a digital format, e.g. 14.0
$race_time = "$race_hour.$race_minute";

my ($current_minute, $current_hour) = (localtime)[1,2];
my $current_time = "$current_hour.$current_minute";

unless ($current_time < $race_time) {
        print "Race time $race_time has passed\n"}
else {print "Collecting prices for race time $race_time\n"};

while ($current_time < $race_time)  {

#wait 15 seconds before taking a snapshot of the market
#this is our example time interval between each set of prices captured

sleep 15;

#let's get the time again now the loop has started, otherwise the loop will never end:

my ($current_minute, $current_hour) = (localtime)[1,2];

$current_time = "$current_hour.$current_minute";

#for testing
print
"$current_time is still less than $race_time, so collecting more market prices now\n";


#-----------now we are ready to collect prices, as per the body of Example 8-2
#-----------write the prices out to a database or a file as shown before

}; #remember to close the while loop outside body of Example 8-2 after writing out prices
   #the program will end when the race time is beyond the current time
```

First, before crafting any program to fetch prices repeatedly we need to decide what we want to achieve in collecting prices repeatedly and adapt the program to suit. Typical considerations might be how many iterations we want to go through, or simply what time intervals we want to specify between capturing prices on a single event. We should

also decide when we want the program to stop capturing prices – e.g. should it continue through a market "in-running" status? Of course we may never stop, in which case the program will simply cease to do anything useful after the event time has passed. If using the Free Access API, there are also practical considerations to consider on service throttling, but we will leave these out of the equation for a second, since our most important consideration is what we actually want to achieve.

There are many ways of attacking the problem of repeating the same loop for a certain frequency and time period, using counters for the number of iterations we want, using the **`sleep`** function to pause between requests for prices (or combining both and counting the cumulative sleep time until a specific total or stopping point), and so on. It is of course possible to retrieve market prices in milliseconds on a single market and repeat ad infinitum, repeating the call as soon as the first call is finished, with no pause.

The maximum frequency with which we can capture prices with no pause in the loop will depend on the bettor's computing set up. In this case, that maximum frequency comprises the elapsed time of the program, including capturing the output, with a GetMarkets call and a GetMarketPricesCompressed call over the internet.

As an indication, setting the market prices loop off repeatedly from the Betwise server took under 0.1 seconds in the timestamp between market iterations, doing so from an older (late 'Nineties) home PC on 2 gigabyte broadband was variable and sometimes just over a second. Whilst sub-second market intervals can be useful for trading activities, this is not the case for picking up trends in market prices as far as betting on outcomes is concerned where a regular sustained frequency is the main asset. Even longer intervals can be useful for identifying trends – and are often more informative as a summary view.

For the purposes of the example, we'll imagine there is a race starting at **2.10 pm**, and our program to monitor the prices for it has been scheduled to execute automatically at **2.00 pm.** Since most market activity increases exponentially in the 10 minutes leading up to the off, the objective of the program is to capture the trending data so that it can be used at any point up until the time that the race starts.

We therefore want to get current market prices and volumes (as per our core routine) at a specified frequency and write these to the database. The prices can be used by any other program in our framework, such as a betting decision making program. Such a program can be running in parallel to our example, querying prices from the database whenever needed whilst the price capture program continues to write prices to the database until it is finished.

First, we assume we have passed in our **marketId** for the 2.10 pm race (automatically reading this from the database or a file on the system), and that our scheduling programs have done their job, ensuring that when the program starts at 2.00 pm, it is the 2.10 pm race that the program considers (we explained how to do this in Chapter 7 *Scheduling – The Key to Automation*).

Additionally, so that the "test condition" for the loop can be specified, we will need to pass in the race time in HH:MM (24 hour format) along with the marketId for the event. The race time is available within the database table **`races`** that we created in Chapter 5, so if incorporating this routine as part of an automated strategy, we would typically select

the market ID and the race time from the database on an automated basis, or select this from a flat file, as discussed for the example automation strategies in Chapter 7.

Then, we start our while loop, to constantly iterate through the core routine already shown in *Example 8-2* to fetch prices. But first we need to do more work to specify the conditions for running the loop.

Since we want the program to stop at 2.10 pm, when the race is due to start, we will have to obtain the system time at every new iteration of the **`while`** loop, and check to see whether or not it is 2.10 pm. The behaviour we want to implement can be paraphrased as, "as long as it is before 2.10 pm, collect prices every *N* seconds". The test condition for the loop is the time being before 2.10 pm. Testing against the current time and performing some action dependent on the test is a generally useful construct for automation of betting strategies within the context of a program and we refer to it elsewhere in the book. However, for the particular objectives of this program, there is, as ever, more than one way to do it, and one way is to change this condition to instead test for the time that the market turns in play (which can of course be rather different from the scheduled off time), as we discuss in the next section, *Using a Market In-Play Test to Fetch Prices Before or After Offtime* after the program walkthrough below.

After the loop conditions have been specified we specify a pause for the frequency of collecting the data, using the sleep statement. The example specifies 15 seconds since we are using the **GetMarkets** call in every loop, which has restrictions on frequency in the Free Access API. (N.B.. Since *Example 8-4* is calling the same market repeatedly, the program could be rewritten to make the **GetMarkets** call before executing the loop, thus executing **GetMarkets** only once, although we would lose any current information returned by that call).

So, before the loop begins let's reformat our 24 hour format race time as a number - so that it can be compared using Perl's numerical comparison operators to the system time. Then we get the system time, also in 24 hour format, and do the same thing – i.e. convert it to a number which can be compared to our race time.

```
my $race_time = "14:10";    #assume a $race_time variable has been read in here
my ($race_hour, $race_minute) = split (/:/, $race_time);

#convert the race time from HH:MM to a digital format, e.g. 14.1
$race_time = "$race_hour.$race_minute";

my ($current_minute, $current_hour) = (localtime)[1,2];
my $current_time = "$current_hour.$current_minute";
```

That gives us the information we need for the first iteration of the loop. Now we execute the loop, according to the test condition in brackets, checking to see whether the system time is yet at the time of the race. When it is, the program will exit. When it's not, we carry on.

Next we get the system time yet again. Since we need to test the loop on the system time for each iteration of the **`while`** loop, this will clobber our previous value and be passed into the test condition on the next iteration. Then we sleep for 15 seconds, giving the market a chance to do something different. This is a completely arbitrary time period. In illiquid markets the market may not move, or the same for "illiquid runners"

i.e. that are unfancied and/or untraded. On the other hand, on the more popular races, on most Saturdays or at big midweek meetings like Royal Ascot or Cheltenham, 15 seconds maybe too long a pause and will mean missing informative tick movements. So the reader can implement their own rules for this, depending on their strategy.

As a rule of thumb, the more data requested and collected, the more maintenance can be expected on both the programs to collect the data and the database itself. In addition, data request charges have recently been introduced by Betfair, which apply in the event of the user making copious calls to the Betfair API (generally relating to a high frequency of calls on the same market). So, the objective is to retrieve a useful amount of data rather than data for its own sake – and that needs to be judged in line with the betting strategy and the likely state of the markets. Testing and refining the strategy is a good way to decide what data is needed, as we discuss in Part 3, *Automated Strategies in Practice*.

```
while ($current_time < $race_time)  {


#wait 15 seconds before taking a snapshot of the market
       sleep 15;

#let's get the time again now the loop has started:
my ($current_minute, $current_hour) = (localtime)[1,2];
$current_time = "$current_hour.$current_minute"


# and collect prices, as per the body of the example function, i.e.:
```

The program now returns to the core example we started with, collecting current market prices using **`get_market_prices_compressed`** and saving them to a database.

```
#       Get market information for this event, and save the array for runner names
```

Next we simply close the new while loop **`};`** before the last lines of the original code to dump the hash of prices to a file, which is now the end of the script. The code creating the hash and dumping it out to a file is now irrelevant, so is deleted (otherwise we would be overwriting the hash key, i.e. the runner name, each time we collect the new set of prices for the runner).

Let's query our database to have a look at some sample output, in this case for a single runner, *Fonthill Road,* whose market data was captured - along with all the other runners in the 15:45 race at York on the 13th October 2007 - using a program based on *Example 8-4*. In the query whose results are shown in *Figure 8-2*, we concentrate on showing the changing prices (together with volumes) for one runner over different points in time. These are all extracted from our **`price`** table, filtered for a single runner, at all timestamp values captured in the last few minutes' run up to a race. So, the MySQL query at the interactive MySQL prompt **`mysql>`** reads as follows:

```
mysql> SELECT runner, timestamp, back, backvol, lay, layvol FROM price WHERE
marketId="20634376" AND runnerId="346969" AND timestamp>="15:41:30";
```

Such a query can also be executed programmatically (and therefore scheduled to run automatically) within the context of a Perl script.

Figure 8-2 shows the output from this particular query. The runner in question, *Fonthill Road*, was the second favourite in the market - and in fact won the race in question.

runner	timestamp	back	backvol	lay	layvol
Fonthill Road	15:41:43	7.80	664	8.00	156
Fonthill Road	15:42:00	7.80	469	8.00	141
Fonthill Road	15:42:16	7.80	433	8.00	36
Fonthill Road	15:42:33	8.00	1508	8.20	165
Fonthill Road	15:42:51	8.20	269	8.40	190
Fonthill Road	15:43:07	8.00	759	8.20	270
Fonthill Road	15:43:32	8.00	517	8.20	453
Fonthill Road	15:43:57	8.20	338	8.40	515
Fonthill Road	15:44:18	8.60	18	8.80	163
Fonthill Road	15:44:35	8.40	404	8.60	311
Fonthill Road	15:44:53	8.20	72	8.40	1476
Fonthill Road	15:45:10	8.00	552	8.20	135
Fonthill Road	15:45:26	7.80	139	8.00	11

Figure 8-2: Market data for a single runner ordered by timestamp

Despite no particular market move in the betting for *Fonthill Road* in the run up to the off, which opened at 6/1 in the betting on course and was returned at a steady (fixed odds) starting price of 6/1, there were still some interesting price movements to fair volume on the betting exchange. We can see movement of nearly a whole point on the back side, from 7.8 to 8.60 and back again in less than 4 minutes.

Using a "market-in-play" test to fetch prices before or after offtime

In the code walkthrough for *Example 8-4* we mentioned that testing to see if the market is in-running or "in-play" is a useful test condition, as an alternative to using the event start time, in order to set up a loop for capturing prices before a race begins.

In fact, there is a drawback with using the offtime as a test condition to capture all prices before or after the actual race begins (as opposed to the scheduled event start). This is due to the frequent delays in starts, especially in horseracing – be it problems with loading the horses into stalls or lining them up over jumps, as well as false starts and longer delays which can push race offtimes out for every race at a meeting.

As well as being an alternative test to use for the type of loop construct used in *Example 8-4*, an "in-play" test is also a test condition that can be used to *begin* a loop for retrieving market prices. In other words, it can be used specifically to identify events that have already started. This can either be to capture in-running prices for further analysis, or to execute a betting strategy that relies on in-running markets. An example might be a strategy for identifying front runners, which may back a contender before the event start and attempt to lay off at a shorter price in running (N.B.. This strategy can also be executed by using bet persistence in suitable markets for the "lay leg" of the bet before the event begins.)

To ensure all prices are captured up until the actual start or that prices captured for in-running markets when the event has started, we can therefore use a test condition

extracted from the Betfair market data already captured. One of the variables we can return from `get_market_prices_compressed` is the "betting delay" indicator used for in-running markets. Before the event starts, this indicator indicates a delay of zero seconds. After the event turns in-play, there is a delay of more than zero seconds before any bet is released to the market. Thus, we can create a variable to use as a test for whether the event has started by extracting this value.

In order to replace the "scheduled event time" test condition given in *Example 8-4* with an "in-play" test condition, we first create a variable, **`$pre_event`** that already tests true (i.e. with a value of 1), stating that the event is not in play (since we will always schedule the program to start before an event). This variable must be declared before the loop starts, effectively replacing the construct we used for determining the current time before, as in:

```
my $pre_event = 1;
```

Next, we start the loop to fetch prices, setting the test condition for the loop according to the **`$pre_event`** variable:

```
        while ($pre_event)  {

        #wait 15 seconds before taking a snapshot of the market
               sleep 15;

        # and collect prices, as per the body of the core example
```

Whilst **`$pre_event`** is true, we collect prices. When it is not true, we stop. When prices are collected on every iteration of the loop, therefore, we must also now test for whether or not the event has started, just as previously we tested to see what the time was on every iteration of the loop.

Thus, after we have made the usual call as follows:

```
my %prices_hash = get_market_prices_compressed($token, $marketId);
```

We also extract the market delay variable that is saved by the subroutine, and use it to set the variable `$pre_event` to its correct value:

```
my $event_delay = $prices_hash{'delay'}
if ($event_delay > 0) {       #i.e. if the market has turned in play
$pre_event = 0};              #the pre_event variable becomes false, and the loop stops
```

For this example, to avoid the possibility that prices are collected after the sleep period, we could explicitly exit the loop on the first occasion that `get_market_prices_compressed` returns an in-play indicator, stating

```
if ($event_delay > 0) {exit};
```

On the other hand, we can also use this construct for the opposite reason to *Example 8-4*, in order to start to collect in-running prices, as opposed to stopping before an event goes in running. In this case, a program to do so could be scheduled to begin at the scheduled race offtime and test for whether or not the market is in-play immediately.

As soon as it becomes in-play, a new variable should be collected, the market status variable **`$prices_hash{'marketStatus'}`**, which returns a status of active, suspended or inactive. Whilst the event is in-running it will be active, and prices should be collected When it becomes inactive or suspended, the price collection program should stop. The criteria chosen to exit the program is up to the bettor's preferred strategy, since in-running markets can come back after suspension in the event of photographs or stewards' enquiries for example.

Price collection can of course be set to continue during an in-play market, but in this case it is worth bearing in mind that for rapidly changing in-running markets, an interval of 15 seconds is too long, so the price collection loop itself should be set to a much higher frequency.

Calculating average price and weight of money

Whilst capturing current market prices enables us to view the current market action, it can give an incomplete picture of trends in the market as a whole. Any one of our automated programs, (such as those collecting multiple prices in the above section) may happen to read prices at a point where the market is shifting. When this happens we may catch a fragment of the last volume available at the previous price, or a price that is available only fleetingly, so that the market information returned is unrepresentative.

Moreover, we need to consider volumes in the market in order to represent what prices are available to "real money". For example, let's say that we wish to place a back bet of **£50**. Currently available volumes to prices are: **£2** at **10.0**, **£2** at **9.0** and **£46** at **8.0**. Clearly our average price is not the average of the prices available **(9.0)** but the average price to real money, or rather the weighted average market price **(8.12)**, which we calculate for each runner within the context of a program in *Example 8-6*.

There are many instances where the average price itself may be directly useful, for example if we are looking to "get on" to a higher volume than is currently available at the head of the market, we will want to know the average price available to real money. Whilst this is rarer for a favourite in a popular market, our automated betting program does not have to be a "high roller" for such data to make sense. In the instance where a market is illiquid, there may be insufficient liquidity at the current price, especially a long time before the off, but nonetheless the average price may be deemed "good value" by the betting strategy in play.

Of course, we can also use the weighted average and total volume information (in this case we are considering the top 3 levels returned by a call to **GetMarketPricesCompressed**) as part of a strategy to determine market movements, for betting or trading strategies. For example we may interpret the total lay volume versus the total back volume, to indicate that there is more "price pressure" on the lay side or the back side of the market, depending on whichever volume is greater. Such an indicator can be used within a betting program to take a position or wait for best price, either to back or lay. In our general betting framework, we would consider such data within the context of the betting decision making stage, as discussed in Chapter 9.

All the information required to create the necessary variables is returned in our **`get_market_prices_compressed`** function, but has not been extracted so far. To keep matters simple, we will rewrite the core example so that, instead of returning the current market values to the database, we calculate the following variables which have been discussed in the background to this section:

- Total back volume
- Total lay volume
- Average back price
- Average lay price

These new variables are written in the example script to a separate database table **`average_price`**, as apart from our previous database table **`price`,** which was for current prices only. Of course, we could also perform all operations, including returning current prices, in one program as opposed to two, and store all the values, including average price and current price in one table if required. Alternatively, the actual data could be stored in the database (i.e. all 3 levels of price and volume information), and the derived data (i.e. average prices and weight of money) could be calculated by any programs accessing the data.

It's worth mentioning before continuing with this example that a call to the newer Betfair API service **GetCompleteMarketPricesCompressed** (introduced in version 6 of the API in 2008) will also return all available levels to back and lay for all runners in the current market. This is a heavier call in terms of returning more data and then having our programs process it, but can be useful where we want to reflect all the liquidity in the market in terms of our average price and volume information (although the downside is that it will be skewed by weight of money left at the extremes of the market, that is unlikely ever to be matched).

Example 8-5: Create Average Prices Table

```
#CREATE average price table for top 3 levels of back and lay prices/ volumes within
AUTODB database

DROP TABLE IF EXISTS average price;
#@ _CREATE_TABLE_
CREATE TABLE average_price
(
        marketId         INT(11) NOT NULL,
        runnerId         INT(11) NOT NULL,
        runner           VARCHAR(25) NOT NULL,
        date             date NOT NULL DEFAULT '0000-00-00',
        timestamp        time NOT NULL,
        av back          DECIMAL(6,2),
        total_back       DECIMAL(10,0),
        av_lay           DECIMAL(6,2),
        total lay        DECIMAL(10,0),
        PRIMARY KEY (marketId, runnerId, timestamp)
);
#@  CREATE TABLE
```

The script to create the *`average_price`* database table can be executed as shown for the other examples creating database tables (in *Example 5-1* and *Example 8-1*) in the *`autodb`* database.

Example 8-6: Retrieve 3 levels of market data, calculate average prices and volumes then save to database

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee 2004-2008.  All rights reserved.                   #
# The code from this example is provided under the terms of the Artistic License 2.0        #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code           #
#-------------------------------------------------------------------------------------------#

#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";           #Free Access API access code

#       other program variables not declared in line
my $back_price;
my $lay_price;
my $lay_vol;
my $back_vol;
my $marketId;
my $timestamp;
my $date;
my $discard;
my %prices;
my %inter_prices;

#       open our database handle for a permanent record of the prices

my $dbh = DBI->connect("DBI:mysql:autodb", "username") or die ("Error:  $DBI::errstr");
#substitute your database and user credentials

#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }

#       For this example we pick one event and type it in, normally this is an
#       automatically created variable
$marketId = "20625238";           #your example event ID

#       Get market information for this event, and save the array for runner names
my @market_array = get_markets($token, $marketId);       #runner_name is key, runner ID
is value
```

```
my %static_runner_data = %{$market_array[0]};
#runner_name is key, runner ID is value

#       Get current market prices
my %prices_hash = get_market_prices_compressed($token, $marketId);
#       n.b. in return hash runner ID is key, runner prices are a value of hashes

$timestamp = $prices_hash{timeStamp};
($date, $timestamp) = split (/T/, $timestamp);
($timestamp, $discard) = split (/Z/, $timestamp);
print $timestamp;
# Data now in memory, calling Betfair services is finished

my @names = keys(%static_runner_data);
foreach my $runner (@names) {

my $runnerId = $static_runner_data{$runner};

#  capture 3 levels of prices and volumes for back and lay markets
my $back_price1 = $prices_hash{prices}->{$runnerId}->{back}->{1}->{price};
my $back_vol1 = $prices_hash{prices}->{$runnerId}->{back}->{1}->{amountAvailable};
my $lay_price1 = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{price};
my $lay_vol1 = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{amountAvailable};

my $back_price2 = $prices_hash{prices}->{$runnerId}->{back}->{2}->{price};
my $back_vol2 = $prices_hash{prices}->{$runnerId}->{back}->{2}->{amountAvailable};
my $lay_price2 = $prices_hash{prices}->{$runnerId}->{lay}->{2}->{price};
my $lay_vol2 = $prices_hash{prices}->{$runnerId}->{lay}->{2}->{amountAvailable};

my $back_price3 = $prices_hash{prices}->{$runnerId}->{back}->{3}->{price};
my $back_vol3 = $prices_hash{prices}->{$runnerId}->{back}->{3}->{amountAvailable};
my $lay_price3 = $prices_hash{prices}->{$runnerId}->{lay}->{3}->{price};
my $lay_vol3 = $prices_hash{prices}->{$runnerId}->{lay}->{3}->{amountAvailable};

#   calculate weighted average prices, back and lay, and total back and lay volume

my $total_back_vol = ($back_vol1 + $back_vol2 + $back_vol3);
my $total_lay_vol = ($lay_vol1 + $lay_vol2 + $lay_vol3);

my $back_multiples =
($back_price1*$back_vol1)+($back_price2*$back_vol2)+($back_price3*$back_vol3);
my $lay_multiples =
($lay_price1*$lay_vol1)+($lay_price2*$lay_vol2)+($lay_price3*$lay_vol3);

my $av_back_price = sprintf ("%.2f", $back_multiples/$total_back_vol);
my $av_lay_price = sprintf ("%.2f", $lay_multiples/$total_lay_vol);


#       test we are getting the right output
print "$runnerId, $av_back_price, $total_back_vol, $av_lay_price, $total_lay_vol\n";

# Deal with saving this information now, first for a permanent record to our database
my $sql = qq(INSERT INTO average_price VALUES
('$marketId', '$runnerId', '$runner', '$date', '$timestamp', '$av_back_price',
'$total_back_vol', '$av_lay_price', '$total_lay_vol') );
my $query = $dbh->prepare($sql);
$query->execute;

#       also create a hash of arrays to represent our runner prices that we subsequently
#       dump to a file:
$inter_prices{$runner} = [$av_back_price, $av_lay_price, $total_lay_vol, $total_back_vol,
$runnerId, $marketId];

}

#       Dump out horse IDs to a file which can be read into our decision making program
#       later

use Data::Dumper;
$Data::Dumper::Purity=1;
open (FILE, '>/home/aeb/inter_prices');
```

```
print FILE Data::Dumper->Dump([\%inter_prices], ['*inter_prices']);
close FILE;
```

Here again, in *Example 8-6*, we have a variation on the core routine that captures current market prices as shown in *Example 8-2*. The script is the same up to and including the point where the Betfair calls to **GetMarkets** and **GetMarketPricesCompressed** are made, differing in the data extracted after the calls are made, and then in what is done with that data to derive further variables of use.

The previous comments made for *Example 8-2* and *Example 8-4* also apply to *Example 8-6* in using a "hard coded" market ID within the script – normally we would let this variable be captured automatically, according to the order of events to be used for a particular betting strategy, as explained in Chapter 7. Any market ID can be chosen and inserted to test the script "as is", however.

As per *Example 8-2* we call the relevant functions to capture the data structure relating to this market. Then, in contrast to *Example 8-2*, we extract data to more levels of depth in the market than previously. So, not only the current back price, volume, lay price and lay volume, but the same for position 2 and position 3 in the market on each side. The current prices and volumes are labelled `$back_price1`**,** and so on, within this script.

The implementation of the algorithm for calculating the weighted average price needs little further explanation, since it is documented within the code itself, in the following lines:

```
#  capture 3 levels of prices and volumes for back and lay markets
#----capture variables, code shown in Example 8-6

#   calculate weighted average prices, back and lay, and total back and lay volume

my $total_back_vol = ($back_vol1 + $back_vol2 + $back_vol3);
my $total_lay_vol = ($lay_vol1 + $lay_vol2 + $lay_vol3);

my $back_multiples =
($back_price1*$back_vol1)+($back_price2*$back_vol2)+($back_price3*$back_vol3);
my $lay_multiples =
($lay_price1*$lay_vol1)+($lay_price2*$lay_vol2)+($lay_price3*$lay_vol3);

my $av_back_price = sprintf ("%.2f", $back_multiples/$total_back_vol);
my $av_lay_price = sprintf ("%.2f", $lay_multiples/$total_lay_vol);
```

Calculating total back volumes and total lay volumes is a simple case of adding all the available volumes together. To calculate averages, we work out how much money can be won and lost on each side of the market, by multiplying the price at each level by the volume available for each level, then adding these figures together for each side of the market and dividing by total money available, for each side of the market, in order to calculate average price.

The weight of money is the difference between the total amounts to back and lay, which can be expressed any way we choose, such as a percentage on the back or the lay side against the total money available. If the back side is a low percentage, we might conclude that the pressure on the market is downward, i.e. the price is likely to shorten. If the reverse and the back side is a high percentage of the total money in the immediate market, then we might conclude the opposite, and that the price is likely to lengthen.

It is important of course to remember that whatever is shown on the back side represents money that has been placed by layers, and whatever exists on the lay side represents money placed by backers. Thus, the pressure of a large amount shown on the lay side of the market compared to a small back amount means there are many people trying to back the horse and few trying to lay it at current prices. Should that situation persist there will be an inevitable decrease in price.

Let's therefore look at the output from running the program, querying the database within an interactive MySQL session.

We run the following query, for the results shown in *Figure 8-3*:

```
mysql> SELECT runner, timestamp, av_back, total_back, av_lay, total_lay FROM
average_price WHERE marketId="20634376" ORDER BY av_back;
```

```
+------------------+-----------+---------+------------+--------+-----------+
| runner           | timestamp | av_back | total_back | av_lay | total_lay |
+------------------+-----------+---------+------------+--------+-----------+
| Dabbers Ridge    | 15:37:56  |    6.31 |       1149 |   7.00 |       412 |
| Fonthill Road    | 15:37:56  |    7.06 |       2100 |   7.82 |       760 |
| Commando Scott   | 15:37:56  |   11.99 |        735 |  13.36 |       127 |
| Tawaassol        | 15:37:56  |   12.78 |        269 |  14.48 |       274 |
| Dhaular Dhar     | 15:37:56  |   14.20 |        633 |  16.21 |       217 |
| Rising Shadow    | 15:37:56  |   16.15 |        224 |  17.75 |       136 |
| Conquest         | 15:37:56  |   16.72 |        264 |  17.93 |       214 |
| Philharmonic     | 15:37:56  |   17.52 |        254 |  19.26 |       129 |
| Zomerlust        | 15:37:56  |   18.47 |        162 |  20.72 |       308 |
| River Falcon     | 15:37:56  |   21.27 |        118 |  23.58 |        84 |
| Hoh Hoh Hoh      | 15:37:56  |   24.30 |         76 |  30.87 |        55 |
| Viking Spirit    | 15:37:56  |   24.85 |        335 |  28.47 |       124 |
| Turnkey          | 15:37:56  |   25.99 |        545 |  31.09 |       159 |
| Obe Brave        | 15:37:56  |   26.77 |        113 |  33.86 |        86 |
| Somnus           | 15:37:56  |   29.02 |        558 |  40.46 |       100 |
| Tournedos        | 15:37:56  |   40.16 |        102 |  48.48 |        37 |
| King Orchisios   | 15:37:56  |   49.61 |        530 |  63.29 |        97 |
| Golden Dixie     | 15:37:56  |   58.92 |         49 |  70.88 |        46 |
| Chicken Soup     | 15:37:56  |   68.76 |         34 |  95.29 |        17 |
| Invincible Force | 15:37:56  |  122.35 |         19 | 150.48 |        38 |
+------------------+-----------+---------+------------+--------+-----------+
```

Figure 8-3: Market data ordered by back price for a single timestamp

A couple of points are worthy of note.

Firstly, the average back and lay prices shown can be compared to the current prices that we displayed earlier for the same race. It can be interesting to compare the two to see if there is underlying weakness in the market for any contender, beneath its current price. In this case, there is no difference in ranking between average and current prices.

Also, we have only one point in time captured for this example. Looking back to capturing current prices for multiple timestamps, we can see that the same can be done for average prices. The method is the same, again within a loop that encloses the block of code that performs the task for a single iteration; however, it contains rules for repeating that block of code at regular intervals.

If we look at *Figure 8-3*, we can see that the highest average amount of money available in the top 3 levels of the market at this particular point in time is for *Fonthill Road* on the back side of the market. The total amount to back is almost twice that of the favourite, Dabbers Ridge, and far more than any of the current lay amounts in the market. Moreover, the total back amount for *Fonthill Road* is almost three times the total lay amount. This indicates upward pressure on the price, with more layers than backers across all 3 levels.

A few minutes later, we can see that the price has indeed gone out, as shown in *Figure 8-2* which shows the results of a query giving a snapshot of prices for *Fonthill Road* alone, taken at regular time intervals. Indeed, if volumes at the current prices are used, we can see that the same pressure from layers also continues in *Figure 8-2*, bearing some relation to price increases, before significant back amounts (shown as the lay volume) bring the price in just before the off.

Summary:
We have looked at looping through multiple events to capture prices, capturing prices at regular intervals before and after an event begins, and extracting varying levels of market information. This has included calculating derived variables such as weight of money and average prices, and in all cases saving retrieved values to a database. All these examples were derived from the same basic routine to make a call to the **GetMarkets** and **GetMarketPricesCompressed** API services for a given market ID.

In general, it should therefore be evident from the examples that price and volume information on a given market at any frequency required can be captured using small programs. The exact combination of variables and the frequency with which programs extract data of course depends on the demands of the betting strategy or subsequent market data analysis required.

Moreover, the final code to capture market information for any strategy can be completely automated for any of the example programs. This is achieved by replacing the `$marketId` variable (or a number of variables, if addressing more than one market at a time) with an automatically generated value assigned to `$marketId`, following the guidelines in the previous Chapter*, Scheduling – The Key to automation*. Once we have a `$marketId` variable automatically supplied for a strategy, we can use looping structures to capture prices on that market at any frequency.

In the next section we look at deriving further variables from raw market data that can also be used to inform later stages of the automatic betting framework.

Calculating Market Overround or Percentage Book

Another key indicator that can influence any betting decision is the current market overround. As we have discussed in Part 1, *Exploring Requirements For Betting Automation,* it follows that the better the price obtained about any contender, win or lose, the more profitable any strategy will be in the long term. If a system (albeit very hypothetical) has produced 1000 selections, each with a price of 2/1, and a strike rate of 33%, or 300 winners, there is a return of 900 points including stake and excluding any

commissions, and a total loss on the system of 100 points. For the same system, all other factors being equal, an increase in the average price to 3/1 should produce a profit of 200 points before any commissions. Price, in this sense, is all that matters for profitable strategies. Whilst this may seem obvious, certain indicators for ensuring that the best price is obtained, such as the market overround, are not always well used.

The market overround, or percentage book, specifically tells us how close the sum of all probabilities (or odds) in the market are to summing up to 100%. If the odds were to sum up to 100%, leaving exchange commission to one side for a moment, we would have a completely fair game, since probability of an outcome where any runner wins the race is of course 100%. The overround therefore refers to the difference between the percentage book and true probability (i.e. 100%) whilst the percentage book refers to the absolute percentage which the market adds up to.

If we are backing, we will seek a market that is as close to, or under 100%, whereas if we are laying we will seek the opposite (also referred to as an overbroke book, in bookmaking or laying terms). Theoretically, a back market under 100% offers the possibility of making a profit whichever contender wins (to varying stakes on each contender, a calculation which is the equivalent of dutching the whole field).

Before getting too excited about dutching any time the market dips below a 100% overround, it should be pointed out that any market rarely stays under 100% for any length of time. Further, there is no guarantee that if we, or rather our automated proxies, do spot such an opportunity that we can get on to the right stakes with every contender. This is due to the fact that:

a) The overround is a price calculation, not a viable volume comparison of prices to the money available, (e.g. so we could dutch a market which was a 98% book, but to do so would require a £20 stake on one of the contenders where only £2 was available to the required price)

b) there is still a time lag, albeit very small, between any “bot” spotting an overbroke market by capturing all prices, and then successfully matching bets to the available volumes on all of those bets. Even the quickest bots can lose out to a market movement that occurs in the milliseconds between completing bet execution and a price changing.

Notwithstanding the above, there are many automated programs that attempt to dutch such markets, although some risk of being left with an unwanted position always remains.

Whilst we may not have the chance to play the perfect book on many occasions, when looking at betting on outcomes, we can nevertheless use the market overround to help with decisions on obtaining best prices for bets we have already decided we want to make.

Of course, the market overround is only one piece of information. It cannot tell us anything about the chances of an individual selection, or whether or not it is a good bet – that is if it is over or under-priced when measured against its ”true” probability of winning.

However, the bottom line, taking the example of backing for an instant, is that any price obtained in a 100% market is likely to be better than a price obtained in a 120% market – a market overround which is not unusual in the traditional bookmaking world. It would be unlikely in such instances that the 20% difference in the book will be uniform across all runners (the current tendency being for prices to be closer between Betfair and bookmakers at the head of the market and more divergent further away), but nonetheless, there is likely to be some price advantage across the board that makes betting generally more favourable, whatever runner has been selected, at such times.

Before looking at Perl code to calculate the overround, we should also bear in mind that the lay market overround (or percentage book) and the back market percentage book are always different due to the existence of spreads between the prices, given that the exchange world presents a different set of probabilities with regard to winning or losing. This is unlike a traditional bookmakers' percentage book, with only one set of prices that are the watermark between the punter and his traditional foe – i.e. there are no price spreads. On the same theme, exchange spreads can be more divergent than one price tick. So, it is important to calculate overrounds for the side of the market that is to be played, depending on whether a back or lay strategy is in operation.

In the code snippet shown in *Example 8-7*, we again assume that we are building upon the basic example program shown in *Example 8-2* from the beginning of this section, this time adding code in order to calculate the overround or percentage book.

First, however, we will need to consider some new variables, and add these to the beginning of the script.

```
#----------------start of script for Example 8-2 not shown

my $percent_book;
my $percent_takeout;
my $lay_percent_takeout;
my $lay_percent_book;
```

After defining other variables to be used in the program and logging in, we call **GetMarkets** followed by **GetMarketPricesCompressed** as per *Example 8-2*, then loop through each of the runner names in the market, also as before. It is during this loop that we will calculate the market overround.

Example 8-7: Adding calculation of market overround within context of other examples

```
#!/usr/bin/perl -w

#       Assume we are looping through each runner as per Example 8-2, 8-3, 8-4 and 8-6
#       All variables in Example 8-2 are available and new variables added as follows:
#my $percent_book;
#my $percent_takeout;
#my $lay_percent_takeout;
#my $lay_percent_book;

#------Calls to Login, GetMarkets, GetMarketPricesCompressed

#       After capturing current (or level 1) prices for each runner, as in:
#foreach my $runner (@names) {

#my $runnerId = $static_runner_data{$runner};
```

```
#$back_price = $prices_hash{prices}->{$runnerId}->{back}->{1}->{price};
#$back_vol = $prices_hash{prices}->{$runnerId}->{back}->{1}->{amountAvailable};
#$lay_price = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{price};
#$lay_vol = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{amountAvailable};

#       Calculate the overround for this contest

$percent_takeout = (100/$back_price);
$lay_percent_takeout = (100/$lay_price);
$percent_book += $percent_takeout;
$lay_percent_book += $lay_percent_takeout;

$percent_takeout = ();                #wipe value of percent takeout for each runner
$lay_percent_takeout = ();            #before moving onto the next runner

}       #end foreach my $runner

#       now do something with the overround values available
#print "OVERROUND = $percent_book\n";
#print "LAY OVERROUND = $lay_percent_book\n";
```

Using the variables for the current market price and volume, our method is to calculate the percentage of the book that each runner "takes out", or in other words what the probability is of the runner winning the race, purely according to the market.

We then store this probability, or `$percent_takeout`, in bookmaker parlance, to its own variable, and in turn add that runner's percent takeout to an ongoing total for `$percent_book`.

```
$percent_book += $percent_takeout;
```

We clear the variable for `$percent_takeout`, before it is assigned a new value by the next iteration of the loop for the next runner, which is in turn added to percent book, until we have been through all runners. In parallel, we also undertake the same procedure for the lay side of the market, saving each runner's takeout to `$lay_percent_takeout` and tracking the total in `$lay_percent_book`.

Once the loop is done, our values for the overround are available in these variables. We can write them to a database table, add them to our hash, or whatever we want, as we have discussed in previous examples. Below, we just show the print statement, to test they have been captured correctly:

```
print "OVERROUND = $percent_book\n";
print "LAY OVERROUND = $lay_percent_book\n";
```

We can also calculate the overround in another part of the betting process, for example within the betting decision making process, using the same method, provided we also read in the price file or database query within that stage of the process, such that we are able to calculate the overround as above.

This statement applies to most examples showing derived variables based upon market information – i.e. we can capture the raw data at this stage, but if that data is made persistent in a file or database, we can decide at which stage of the betting process we create and use derived variables. As data items they belong to the market information stage, since they can be calculated based on all information we collect in these phases. However, the use of such variables as the overround, weight of money, average price and so on (at least within a betting strategy, as opposed to research) is in the additional

information they provide to the betting decision making process. So that is where they are used.

Analysing the entire price ladder per runner

Using the **GetMarketPricesCompressed** API service gets us so far in analysis of the current market but to be thorough about the market for each runner we should look to the entire trading history for each horse (which, aggregated, of course gives us the entire market history) as well as all the outstanding offers for the horse on the back and lay side, at all price levels.

GetMarketTradedVolume and **GetDetailAvailableMktDepth** are the services that can be used to retrieve this information, equating, in terms of the website User Interface, to the *Traded and Available* table for each runner.

The newer function **GetCompleteMarketPricesCompressed** can also be used to retrieve all current liquidity information, (just as its namesake, **GetMarketPricesCompressed** does for the first 3 levels to back and lay in the market) for all runners in the market. This can avoid the need to call **GetDetailAvailableMktDepth** for each runner, if wishing to analyse the price ladder for all runners at the same time. Similarly, a single **GetCompleteMarketTradedVolume** API function will in future be able to retrieve traded volumes at all levels for every runner, avoiding the need to call **GetMarketTradedVolume** for each if the required analysis is over the whole market – although the latter function remains a viable solution.

We refer to a "price ladder", since that is what the table shows us for each runner, and the type of interface which is also adopted by trading software. Within the Betfair user interface, the table is simply a static report. An example table is shown below for those unfamiliar with this data. In terms of the user interface, the table is shown alongside some other market information we have already covered in the **GetMarketPricesCompressed** overview, including the summary market information for the runner on the left and **Price/Volume over time** graphic.

However, we are interested in the table represented by the columns on the right hand side of *Figure 8-4, Traded and Available*:

Betting on: Best Alibi

Total matched on this event: **£559,798**

Reduction Factor **28.5%**

Betting summary - Volume: **£281,766**

Last price matched: **3.85**

Traded and Available			
Odds	To back	To lay	Traded
1.01	£1,830		
1.04	£27		
1.05	£12		
1.06	£15		
1.10	£33		
1.19	£853		
1.20	£5		

Inverse Axis

The information on this page may be slightly delayed.

1.39	£154		
1.50	£10		
1.65	£5		
1.76	£5		
1.96	£104		
1.99	£800		
2.00	£2		
2.06			
2.10	£273		
2.12	£43		
2.14	£614		
2.16	£1,585		
2.18	£339		
2.20	£17		
2.24	£202		
2.26	£5		
2.28	£1		
2.30			£1
2.38	£2		
2.40	£574		
2.46			£55
2.48			£70
2.50			£370
2.52	£94		£449
2.54	£1		£332
2.56			£250
2.58	£3		£82
2.60	£50		£562
2.62	£187		£94
2.64			£120
2.66	£2		£557
2.68			£286
2.70	£159		£535
2.72			£80
2.74	£187		£381
2.76	£170		£121
2.78	£32		£60
2.80	£15		£723
2.82			£471
2.84	£9		£1,274
2.86	£100		£1,337
2.88	£196		£2,383
2.90	£10		£5,825
2.92	£100		£6,117
2.94	£51		£6,884
2.96	£48		£6,505

2.98	£37		£5,990
3.00	£616		£14,873
3.05	£103		£5,936
3.10			£2,467
3.15	£350		£1,343
3.20	£50		£499
3.25	£793		£422
3.30	£125		£169
3.35	£122		£424
3.40	£475		£1,976
3.45	£47		£2,277
3.50	£794		£1,807
3.55	£1,151		£6,503
3.60	£1,082		£11,515
3.65	£61		£19,814
3.70	£579		£27,965
3.75	£761		£19,596
3.80	£909		£11,733
3.85		£729	£11,799
3.90		£1,048	£6,327
3.95		£1,450	£7,098
4.00		£1,866	£24,438
4.10		£4,771	£29,487
4.20		£3,931	£19,945
4.30		£587	£11,309
4.40		£452	£129
4.50		£789	
4.60		£371	
4.70		£27	
4.80		£20	
4.90		£16	
5.00		£113	
5.10		£83	
5.30		£23	
5.50		£20	
6.00		£12	
7.00		£1	
11.00		£20	
90.00		£3	
1,000.00		£167	

Figure 8-4: Traded and Available Prices/Volume on Betfair, corresponding to GetTradedMarketVolume and GetDetailAvailableMktDepth API functions

.

There are 2 calls to cover this data in terms of using the API, and each call is made for a particular runner in a market, rather than for returning the market as a whole.

Dealing with each of the 4 columns in turn, column 1 gives us the prices for amounts matched and available on this horse, column 2 shows us what is available at each price in the current market to back, column 3 what is available in the current market to lay, and column 4 the total volume traded at each price.

.

What can we do with this information in automated programs? We can use it as we earlier discussed for other indicators, such as weight of money, average price and market overround (although the prices for **GetDetailAvailableMktDepth** are cached, so do not give the same live picture as **GetMarketPricesCompressed**, but do give greater context in terms of the entire pricing ladder), to help us determine automatically either which way the market will move for a particular runner, or what odds offer the best potential for being matched.

Note that for Betfair SP markets, two additional columns are added to represent bets placed in the SP markets before the off, being **Backers' stake** and **Layers' liability**. This data can also be retrieved within the calls mentioned, as we show in *Example 8-8* for all levels of traded volume

In one respect, the data allows us to calculate the same type of statistics we used earlier, but for all levels of depth. This is what we can achieve with the data retrieved from **GetDetailAvailableMktDepth**, although since this is more detailed offer information, this is cached every 20 seconds. Additionally, we can see more clearly at which price levels an opportunity might exist at which to place speculative offers. In the market shown in *Figure 8-4*, for example, we can see that there is no offer available to back at a price of 3.10, despite the fact that over £2000 has been matched previously at this amount, and there is over £10000 waiting in the market to be matched at lower prices. So there is an opportunity to be "first in the queue" at that price.

There are a few further useful differences in the data available in the price ladder as opposed to the data we were dealing with in the previous section, and it is these that we will focus on. Most important is that we are not dealing only with offers, but with how the market has actually behaved, as represented by the volume traded, as returned by the service **GetMarketTradedVolume**, the Perl function for which is also available in our library of functions.

Offer information in the market is all well and good, but only traded volumes are the equivalent of punters "putting their money where their mouth is". Making any large offer to be matched at a level well away from the current market (to take the extreme case, at odds of 1.01 or 1000) does not generally tell us anything about the market or the level of market confidence in any particular contender at any point in time. Since any amount, available at any price other than the current market, may be put up and removed at whim, it is questionable to what extent we should use any information that is far removed from current levels of activity in order to predict price movements. We can rely on traded volume in the exchange markets, however, since we know that market participants have traded at whatever prices and volume levels are returned by this function.

Therefore, we can use **GetMarketTradedVolume** to calculate a number of statistics from the traded volumes rather than the offers, which may help us better understand and predict the market. For example, we can calculate the weighted average price, as we showed for the available 3 levels of offers in the market in the previous section, but this time for all the amounts in the historically traded market. We refer to this value as the *average traded price*, representing the average for all prices, at their respective volumes traded, which have been matched. In one sense, this is the most representative price as to any runner's chance in the market, since the average price represents all risk that has been taken. In addition, we can retrieve the highest and lowest prices at which runners have been traded to date, in order to give ourselves a handle on the trading range for any given runner.

The example code below shows the calculation of these values from the **GetMarketTradedVolume** function, shown in the context of adapting the core routine for extracting market prices by **marketId**, *Example 8-2*, with which we started the Chapter. This time we go further than showing an example which demonstrates one call only, to show how we can build up a program using functionality we have already covered. We also generate a report to show the output rather than saving it to a database or creating a data structure within a flat file (as per options covered in previous examples). The function call for **GetMarketTradedVolume** is designed to be used for one runner; but in *Example 8-8* we calculate average prices for all runners in the market, and return those with other market information for each runner.

The objective is to create a report for each runner in the race listing:

- Current back price
- Current lay price
- Calculated average price based on all trading history
- The lowest price at which each selection has traded
- The highest price at which each selection has traded.

Example 8-8: Producing runner price statistics with market traded volumes

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee 2004-2008.  All rights reserved.             #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#

#       Objective of script:
#       Fetch market prices for a given Betfair market ID,
#       Fetch all traded volume information for each runner in the market;
#       Calculate average traded price, lowest odds matched and highest odds matched
#       for each runner in the market;
#       Print out the variables captured to a report or save in persistent data structure

#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
```

```
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";           #Free Access API access code

#       other program variables not declared in line
my $back_price;
my $lay_price;
my $lay_vol;
my $back_vol;
my $marketId;
my $timestamp;
my $date;
my $discard;
my %odds_array;

#       open our database handle for a permanent record of the prices
open (REPORT, ">/home/aeb/runner_price_statistics_report");


#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }


#       For this example we pick one event and type it in, normally this is an
#       automatically created variable
$marketId = "20625238";           #your example event ID

#       Get market information for this event, and save the array for runner names
my @market_array = get_markets($token, $marketId);       #runner_name is key, runner ID
is value
my %static_runner_data = %{$market_array[0]};
#runner_name is key, runner ID is value

#      Get current market prices
#       runner ID is key, runner prices are a value of hashes

my %prices_hash = get_market_prices_compressed($token, $marketId);
$timestamp = $prices_hash{timeStamp};
($date, $timestamp) = split (/T/, $timestamp);
($timestamp, $discard) = split (/Z/, $timestamp);
print "$timestamp\n";
# N.B.:  hashes for static data and current market prices are now in memory

my @names = keys(%static_runner_data);
foreach my $runner (@names) {

my $runnerId = $static_runner_data{$runner};

#print "$runnerId\n";
my $back_price = $prices_hash{prices}->{$runnerId}->{back}->{1}->{price};
```

```
my $lay_price = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{price};
my $back_amount = $prices_hash{prices}->{$runnerId}->{back}->{1}->{amountAvailable};
my $lay_amount = $prices_hash{prices}->{$runnerId}->{lay}->{1}->{amountAvailable};


#capture all trading history for the runner

my %traded_hash = get_market_traded_volume($token, $marketId, $runnerId);
my $traded_array = $traded_hash{volArray};

#print Dumper ($traded_array);


#declare variables that will be used outside the loop
my $total_vol;
my $total_amount;
my @odds;

foreach my $traded_vol (@ {$traded_array} ) {

my $amount = $traded_vol->{totalMatchedAmount}{content};
my $odds = $traded_vol->{odds}{content};
my $vol = $amount * $odds;

#any bets requested "blind" at starting price can be picked up as follows
#to pick up all bsp bets a running total should be kep as per $total_vol
#my $bsp_liability = $traded_vol->{totalBspLiabilityMatchedAmount}{content};
#my $bsp_back = $traded_vol->{totalBspBackMatchedAmount}{content};

push @odds, $odds;

#print "$amount\n";
#print "$odds\n";

$total_vol += $vol;
$total_amount += $amount;

}       #end foreach traded_vol for runner name

#print "$total_vol\n";
#print "$total_amount\n";

my $lowest_traded = $odds[0];
my $highest_traded = $odds[-1];
my $av_traded_price = sprintf ("%.2f", $total_vol/$total_amount);

#my $traded_timestamp = $traded_hash{timeStamp};
#print "$traded_timestamp\n";

#print "$runner, $back_price, $lay_price, $av_traded_price, $lowest_traded,
$highest_traded\n";
$odds_array{$runner} = [$back_price, $lay_price, $av_traded_price, $lowest_traded,
$highest_traded];

} #end foreach runner in market loop

#create an array of runners ranked by back price (i.e. the first value in the odds_array)

my @ranked_runners =
sort { $odds_array{$a}->[0] <=> $odds_array{$b}->[0] } keys %odds_array;

foreach my $ranked_runner (@ranked_runners) {

#print "@{ $odds_array{$ranked_runner} }\n";     #test

my ($back_price, $lay_price, $av_traded_price, $lowest_traded, $highest_traded) =
@{ $odds_array{$ranked_runner} };

format REPORT_TOP =
RUNNER            BACK  LAY   AVG   LOW   HIGH
.
```

```
format REPORT =
@<<<<<<<<<<<<<<<  @<<<  @<<<  @<<<  @<<<  @<<<
$ranked_runner, $back_price, $lay_price, $av_traded_price, $lowest_traded,
$highest_traded
.

write(REPORT);

}
```

The script starts much like our others, with the exception of a couple of new variables to be used at the start of the program. The point at which it starts to vary from our core example for fetching prices for a given market is after looping through the usual array of market runner names, converting each to its corresponding runner ID, as below:

```
my $runnerId = $static_runner_data{$runner};
```

At this point, as before, we extract all the values we want from **GetMarketPricesCompressed** in terms of currently offered price and volume for each runner, and then start a new loop with runner ID, session token and market ID supplied as arguments to **GetMarketTradedVolume**, in order to extract and calculate traded volumes for each runner.

Now we have a loop running within a loop, since each time we take the market runner name, calling the **GetMarketTradedVolume** for that runner, as well as extracting the last called values for current prices.

Here's how that second loop works:

```
foreach my $traded_vol (@ {$traded_array} ) {

my $amount = $traded_vol->{totalMatchedAmount}{content};
my $odds = $traded_vol->{odds}{content};
my $vol = $amount * $odds;
```

The variable **`$traded_array`** is a value from the hash returned by our function call. This value is itself a reference to an array; so first we dereference that array in order to loop through it.

The array itself is a list of all price levels traded, from the lowest to the highest price, and the volumes that were traded for each of those levels. We therefore access two useful elements of information for each level – the **`$amount`** traded and the price, or **`$odds`** matched for that volume.

Note that to find any bets specified for starting price we can extract those values too – by extracting the variables that are commented out below

```
#my $bsp_liability = $traded_vol->{totalBspLiabilityMatchedAmount}{content};
#my $bsp_back = $traded_vol->{totalBspBackMatchedAmount}{content};
```

However, given that we are calculating the average price matched at the time of running the function, and that BSP bets are only matched to any price once the BSP takes effect at the beginning of a race, we can safely leave these from our calculation.

To calculate our average, we do the same as with our earlier example of offers, working out the volume multiplied by price for every price level, then keeping a running total of that number, as well as the total volume traded:

```
my $vol = $amount * $odds;
$total_vol += $vol;
$total_amount += $amount;
```

After the loop for this particular runner has finished, and before iterating the loop for any remaining runners, we take the traded volume variables created for the runner and calculate the average price, using the sprintf function to format it to 2 decimal places.

```
}       #end foreach traded_vol for runner name

my $av_traded_price = sprintf ("%.2f", $total_vol/$total_amount);
```

Another variable that is created in the example script and is worthy of comment is an array for all the odds amounts at which trades have been matched. As we go through each array element, we add each set of odds traded to this array **@odds**, as in:

```
my $odds = $traded_vol->{odds}{content};
push @odds, $odds;
```

The array persists when the loop to extract traded values for each runner has finished, since we declared the array before the loop began. Thus, once all the values have been added within the foreach loop, **@odds** gives us the facility to extract a number of useful statistics about the trading range of this particular runner. Here, we simply take the highest and lowest values.

```
}       #end foreach traded_vol for runner name

my $lowest_traded = $odds[0];
my $highest_traded = $odds[-1];
```

The trading range is simply the difference between the two values. However, it can often be that the highest traded value and lowest traded values are outliers which are unrepresentative of the real trading range, being matched to small money, early in a market, or simply by user error (e.g. matched trades at 1.01 before an event). Therefore a more useful value can instead be the second and penultimate values for the array (or some other rank of choice), as in

```
my $pen_lowest_traded = $odds[1];
```

…or we can calculate instead values for a high and low range using a minimum volume (e.g. calculating the average price of the lowest and highest matched amounts to at least £100). All this will help determine whether a market price is comfortably within its trading range, or pushing new boundaries, which itself can be a useful price indicator.

Finally, before closing the loop for each runner, we push all the price values that have been calculated to a new data structure, (a hash of anonymous arrays), so that we can subsequently sort the data structure by a price ranking, and extract any values to write to a report.

```
$odds_array{$runner} = [$back_price, $lay_price, $av_traded_price, $lowest_traded,
$highest_traded];
```

```
} #end foreach runner in market loop
```

This ends the main section of the program, the next part deals with sorting the data and writing it out to a report.

Previously we have written the data structure to a flat file which can be picked up for reuse by a subsequent program, or written values to a database for a more persistent record. There is no special reason for writing the values out in a report format below, apart from the fact this can also be another useful output type that has not yet been touched upon. In an actual betting strategy, we would typically write out to a database for persistence as we have already shown, and/or reuse the data structures created within the program for the next part of a betting strategy (e.g. making a decision to back or lay based upon price).
In practice, a betting strategy using the **GetMarketTradedVolume** service is likely to focus on a few selections at the head of the market, where liquidity is highest. Thus we may well prefer to use it for a subset of selections within an event. The subset can be chosen by ranking contenders by order of market price, as discussed for the MySQL query shown in *Figure 8-1*, or, within the context of a data structure in program memory, by sorting market prices as an array, then taking the first element(s) of that array as the top of the market.

We perform such a sort (i.e. within a program context) in the next line of *Example 8-8*, in this case purely to rank the runners by back price for more logical display within the context of the report:

```
my @ranked_runners =
sort { $odds_array{$a}->[0] <=> $odds_array{$b}->[0] } keys %odds_array;
```

This line of code sorts all the keys of our data structure, which are of course names of runners, by order of back price. Normally in a hash, it would be enough to sort by values, so `$odds_array{$a}` with `$odds_array{$b}`. Here, since we have stored an anonymous array as the value of each key, we have to go further and specify the first value of that array (i.e. back price) by referring to the relevant element of the array, i.e. `$odds_array{$a}->[0]`.

Lastly we loop through `@ranked_runners`, printing all values out to a file, whose handle is `REPORT`, for subsequent inspection. Perl lets us specify a simple format for spacing the printed variables and creating a title line – we could of course make this much more sophisticated with a little programming if needed.

In this final example we can see how it is possible to start bringing a number of different price variables, which are pertinent to each runner, together in one place. This can create quite a rich picture of the overall market and its trends, any or all of which can be used in an automated environment.

As with the other examples from this chapter we can make any number of enhancements or adaptations, depending on the use required for the data structures within the context of a betting strategy.

For example, we make the call for getting current market prices only once before the loop begins, although we could also make a new call to **GetMarketPricesCompressed** every time we call **GetMarketTradedVolume**. This would enable us to be very precise about comparing current market prices with average traded prices for each timestamp - which we might want to do if executing trades based upon differences in these variables. In the run up to the off, the traded history can be a lot of data for each runner, so these calls take longer than others (thus creating a potential lag between the first current market price obtained and the last traded history data captured). However, this lag is still in the region of milliseconds, so not significant enough to distort the general trend or for the purposes of the example report.

To repeat the call for regular time intervals (thus capturing the data required to create a moving average and other indicators), as well as to choose the correct event automatically, refer to the earlier sections of this chapter on extending the program, as well as the previous chapter on automatic scheduling.

Summary:

Having looked at how market data can be captured iteratively and how to derive variables for a single point in time (such as average traded price and weight of money), it should be clear that many possibilities exist to amalgamate the capture of variables with the frequency of capture to create moving averages, trading envelopes and so on. We can imagine producing and working with many more derived variables, and analysing the relationships between them, which will take us firmly down the route of producing indicators for trading strategies.

Indeed, when looking at market information alone, there are clear parallels with the analysis of financial market data, which opens up a whole range of additional possibilities to explore in terms of market analysis and price prediction.

Such analysis, although entirely market oriented, can be useful to help determine market confidence in a contender which may be predictive of its chance, as well as short term price movements. In bookmaker terms we have a “steamer”, which is a very obvious observation of a horse’s odds tumbling, or a drifter (the reverse). In exchange markets at one point, backing all drifters was more profitable than the reverse, although that does not indicate that drifters win more, simply that the price determines long term profitability. In the exchange markets, as well as clear price movements for steamers and drifters, we also have subtle, shorter term levels of information, from which the trader hopes to take advantage, with price movements in either direction, determined by weight of money in the market at any particular time.

As we observe elsewhere, betting and trading strategies overlap in the world of the exchange, where at the extremes we have single bets on outcomes versus iterative trading on price movements. Whilst our emphasis is on automating the betting process, rather than trading per se, it is important to note that in this overlap, the same functions are used, so can be pursued for creating trading strategies, and there are lots of useful possibilities enabled by an exchange where the edges blur. These possibilities create the opportunity to hedge and profit where market information can be used in conjunction with a view on the fundamental chance of each contender.

Chapter 9: Automating Betting Decisions

Before taking an automated betting decision, as with an interactive decision, we require a selection or a method of assessing winning chances as well as data inputs taken from the market. Whilst we can't document every type of betting decision, and by implication discuss every possible betting strategy, we can highlight some of the key characteristics and the ways that manual decisions can be automatically implemented.

At a high level, this part of our automated framework draws together any information that is relevant to the strategy and applies the rules of the betting strategy to that data. The information supplied to the decision program can include:

- The details of the event (and contenders) from a horseracing database or other data format
- An assessment of each contender's chances in any given event, translated into tissue prices OR
- A selection made automatically at the race analysis stage or captured from a third party source
- The current set of market prices and volumes to back and to lay (to any level of market depth – from first in queue onwards)
- Price indicators based on historic price, trade and volume information for each contender such as average traded price.
- Previous results from the days' racing for the purposes of updating variables such as going and draw bias
- Previous results from the betting strategy during the day to inform staking plans such as stop/ loss
- Account or betting bank information for the purposes of automatic staking.

Aside from the overall method in the betting strategy we must determine optimal execution, which will include timing in visiting the market, and the frequency with which we visit the market. Of course, a betting strategy may be designed in order to explicitly take advantage of anomalies or fluctuations in the market, as with a strategy that involves elements of trading or hedging bets.

Using the oddsline that we created in Chapter 6 as an example for input, we will illustrate some of the aspects involved in implementing a decision making process, since many different strategies, and therefore decision criteria, are possible when betting an oddsline.

Example betting decisions based on an oddsline

In this section we consider the process for basing betting decisions upon an oddsline such as the one we generated as an example in Chapter 6*, Assessing Contenders* and discussed in the context of an example strategy for automation in Chapter 7, *Scheduling – The Key to Automation*.

Of course, creating an oddsline in itself is not a betting strategy, but simply a means of representing an assessment of winning chances. Each oddsline will differ depending on the methods used for assessment, and for calculating chances based upon that assessment. However, the key operations required with regard to betting an oddsline at this stage of the process – namely taking an oddsline as an input from an external file, adjusting for non-runners, and comparing the oddsline to the market – are general for any oddsline strategy.

The general principle is based around comparing oddsline tissue prices with market prices and looking for "overlays" – US racing parlance for horses whose odds are higher than their true price – i.e. for which there are too many layers in the market. These are the kind of bets we also hear referred to as "value" selections for the same reasons. Although the word "value" has become something of a cliché in racing circles for justifying a bet, this is usually because there is no quantification of how value is determined; here that is not the case. We should be able to pull every aspect of an automated strategy apart and quantify it. Equally, an oddsline can be used to identify contenders who are "overbet", or "poor value", in order to determine lay bets.

To do this automatically we therefore require an oddsline which has been automatically generated by the bettor or been supplied by a third party; in terms of logistics the parentage of the oddsline is of less importance than that it is available as a clean input file to our decision making program.

The oddsline will represent winning chances for each of the declared runners in each qualifying race on the day. The betting strategy will determine what type of races are qualifiers and on what basis the tissue prices in the oddsline are calculated. The oddsline methods are the "secret sauce" of the strategy, as we discussed in Chapter 6; at the decision making stage, the question is on what basis would we like to use that information to create a profit.

The example oddsline we generated in Chapter 6 was based on the Racing Post's Postdata table; every race in the UK each day is a qualifying race, and oddslines are produced for all - in reality a betting strategy might be more selective. Since only qualifying races will be written to the oddsline file, then whatever exists in the input file is what is presented to the program for consideration - in this case that is all races; again, considering the widest case first means we have more scope for possible improvements later.

Comparing an Oddsline with the Market; Correcting for Non-Runners

Our decision making process will find the oddsline for any given race 5 minutes before the race is due to start. Automating execution will be subject to an appropriate scheduling process such as described for this strategy in Chapter 7*, Scheduling – The Key to Automation.*

The first step after retrieving the appropriate race oddsline is to work out whether there are any non-runners in the race. To be precise, we need to determine whether there is any difference between the declared contenders represented in the oddsline and the actual contenders at the time we wish to bet. So, our decision making program goes

to the relevant market for the race and captures the names of those runners who are going to post (metaphorically speaking, but that's more or less where they should be 5 minutes before a race) together with their prices. We compare each runner's name in the oddsline with the corresponding runner name in the market, and if there is no match we know it is a non-runner. Otherwise, if every runner is matched, we know that all go to post – at least this is true at the time we make the decision; afterwards, the price of any contender which we bet in a race that is affected by non-runners will be taken care of by a reduction factor automatically applied to our odds by the exchange.

However, if there are non-runners before we have made a betting decision, we need to make a reduction in the oddsline equivalent to the price of the removed runners. This is the bookmaking or betting exchange equivalent of making reductions to prices that are already taken on contenders when the field size is subsequently reduced.

Since we calculated all the tissue prices in the oddsline on the basis of declared runners to a 100% book, we will need to reduce the prices of the remaining runners to reflect the fact there is now a greater probability that any of the contenders will win – i.e. maintain a 100% book for the remaining contenders. We thus work out the overround based on the tissue prices of the remaining runners, then as a "quick fix" work out the factor needed to increase the new overround to a 100% book. Finally we apply that factor to the tissue price of all runners in the race. We will therefore expect that if there are any non-runners, the percentage chance of each runner who remains will increase, and the odds will decrease (i.e. move closer to 1). N.B. It would also be possible to calculate the oddsline on the basis of actual runners going to post rather than in advance, however splitting up the tasks is more convenient for the examples.

Basing betting decisions on the oddsline

Firstly, we will consider the rules applied to the oddsline used for our example strategy, then look at some of the possibilities for basing betting decisions on oddslines in general.

For our example oddsline, having adjusted for any non-runners, we ensure that the oddsline is ranked from those contenders with the greatest to the lowest chance (i.e. our tissue price favourite is the first in the list). Then we take the top third of the field in terms of this order. Thus, we will divide the field size by three (rounding down as opposed to up in the event of a resulting number of runners that is not an integer). Whatever number is produced will be the top *N* contenders to consider betting from the oddsline ranking (eg, in a field of 12, 13, and 14 runners we will consider the top 4 horses).

Next, we compare each oddsline tissue price for that runner with the market price, with the objective being to look for overlays, or horses that are have been made too long in price by the market compared to their real chance.

At this point in determining rules for the decision making process we are dependent on the logic behind any betting strategy, which is bound to the input data. In the case of our example the input data is an oddsline which is underpinned by Postdata variables. The logic of our betting strategy in limiting choice to the top third only is to take the most

strongly rated horses and only consider those for backing. We could do the same with overbet horses, and create a laying strategy. We could take the top 3, or the top 5 in the oddsline and look for overlays or overbet horses, regardless of field size. Or only the top rated, or all horses in the oddsline. Or any of these combinations, coupled with the application of a percentage “value buffer”, in case of error.

There really are any number of ways to play an oddsline from a decision making point of view – which avenues we pursue and whether we pursue them depends on our confidence in the oddsline and the way it was put together. So it is in the case of the example we are using. The reason that we do not look at overbet horses in our strategy, or apply the oddsline any further than the top third in the field, is all down to our example input. The Postdata scores used to build it explicitly state strengths for each contender but in many cases leave weaknesses open to question – for example any question mark or blank value is translated to zero.

A question mark or a blank is fine if we are considering a race manually, it indicates we may have to do further research in order to form an opinion, but in the case of producing an automated score any such contender will receive a zero for this attribute. Does that mean we should have confidence in laying it? By definition, we can be more confident that a contender is well suited to the conditions of the race if it appears near the top of the oddsline, since we know that in most cases any contenders near the top of the oddsline have been explicitly rated with positive scores for each attribute. In the middle or bottom of the ranking, where a bunch of horses may have negative attributes or attributes that are simply unknown, differentiation can be uncertain. This also accounts for the reason that we take the top third of the oddsline, in our Postdata produced example, as opposed to a fixed number, such as the top 5 in the field. In smaller fields, a fixed size would mean us dabbling automatically with selections towards the middle or bottom of the rankings, where we have more doubts about the oddsline’s predictability.

By the same token, there is a potential weakness in that we will bet more in races with big fields than small fields, and maintain level stakes on all bets. Given the number of contenders we are considering is related to field size, we therefore risk more. This is a deliberate part of our strategy, however, in that the prices of longshots on Betfair, is at the time of writing where the biggest overlays are to be found. So the strategy is somewhat biased towards obtaining big prices on longer shots which we think we have a hope of significantly beating SP and getting a bigger price than their true chance. By the same token, we still limit to the top third of the field, so are avoiding complete “no-hopers”.

There are many weaknesses in the example oddsline based on the above, an obvious one being that it does better rating horses with lots of known form (so for example a horse who has never encountered a certain going type will score lower though not necessarily be inferior than one who has performed on it). In this sense, the oddsline is simply betting on known form over unknown form in many race types. It is not a disastrous strategy, in any case, to go with contenders with known form over unknown form in any event, but it might be foolish to base a universal strategy around it. On the other hand, in races where the majority of horses are exposed and tried in many conditions, such as is often the case in handicap races, the oddsline can be seen as more interesting.

For more universal applications of an oddsline strategy – going further down the rankings, laying etc - we will do better with a more comprehensive backtesting approach to producing the oddsline, and we look at using such examples in Part 3, *Automated Strategies in Practice*.

So, it is clear we can play in many ways around an oddsline. In this case, having chosen the top third of the field, if the oddsline price on a top ranked contender is lower than the market price, (that is to say our oddsline makes the percentage chance of winning higher than the market does), the betting decision making program will save the runner details to place a bet on it.

After we have assessed our runners against the market prices available and saved those that we wish to bet on, we simply bet them. We have to stipulate event details, runner details, stake, required price and whether a back or lay bet. The variables that most concern us within the context of applying the betting strategy logic are the stake and the back price.

We could apply any staking plan we wished at this point, or we could leave it to the next stage of the framework (i.e. betting execution); the same applies to obtaining the required price. Many possibilities are open for stipulating different prices and automatically calculating stakes based on available bank size, or using other algorithms. In the standard example we will simply take the currently available back price if it offers better value than the tissue price, and stipulate a flat, level stake with regard to the staking plan. Taking a value price and using a level stake are both sensible options for any betting strategy, although the level stake here is within the context of backing a variable number of horses per race, and is thus a dutching strategy. We discuss this further later in the chapter.

What about timing - do we play the market as it is or wait awhile? For our standard example, we simply take the current price inputs (from the ***inter_prices*** file created by supplying a **marketId** automatically to *Example 8-2*) and specify the last back price captured (i.e. stored within that file). By the same token, any of the price indicators discussed in Chapter 8, *Capturing and Using Market Information,* could be inputs at this stage and a decision about whether or not to take current market prices based upon those (e.g. to only take the current price if the market overround is within a certain range). Alternatively, we can program our strategy to take a higher incremental price in relation to current prices, as we show in *Example 9-2*.

In general, however, asking for the back price in the last 5 minutes before a race time usually guarantees a good overround, and, given the time lapse in obtaining the price and a computer program executing, we are also pretty sure to get on, as we see when we review live results in Part 3, *Automated Betting Strategies in Practice*.

Finally, we write out all the details necessary for executing the bet, so that it can be picked up by the betting execution program discussed in the next chapter. Splitting the programs into their atomic parts enables us to test the decision making strategy to ensure we are writing out the correct bet details before pressing ahead and creating a production version which can amalgamate both parts of the process, if required.

Example Oddsline Decision Program

In this section we walk through an example betting decision program ***bet_formation.pl***. The basis of the strategy for the example is described in the previous section. The code shown in this section implements that logic, and can be adapted to any oddsline that we could use as an input file.

For the betting decision process, we require an example oddsline file, here referred to as ***formatted_oddsline*** which itself contains times and names of courses for each contender. This oddsline file is the one first discussed in Chapter 6, *Assessing Contenders*.

Example 9-1: Selecting bets based on an oddsline (bet_formation.pl)

```
#!/usr/bin/perl -w
use strict;

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.                  #
# The code from this example is provided under the terms of the Artistic License 2.0        #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code           #
#-------------------------------------------------------------------------------------------#

#       input oddsline
open (ODDSLINE, '/home/aeb/formatted_oddsline');

#       write to a report to show what has happened
open (REPORT, '>>/home/aeb/betting_report');

#       output any selections to bet
open (SELECTIONS, '>/home/aeb/selections'); #only converts Postdata ratings

#declare variables
my $percent_takeout;
my $total_overround;
my $runner;
my $runners;
my $total_takeout;
my @ranking;
my %final_tissues;
my %tissues;
my %inter_prices;
my %bfair_odds;
my @horse;
my @horses;

print REPORT "################   NEW ODDSLINE  ##################\n";

#       read in market information (captured in an earlier program
{
open (PRICES, '/home/aeb/inter_prices');
local $/;
eval <PRICES>;
close PRICES;
}

#       Read in the current event and write out events schedule ready for the next event
open (EVENTS, "< /home/aeb/events_schedule");
open (TEMP, "> /home/aeb/events_temp");
my @events = <EVENTS>;
my $event_details = shift(@events);
foreach my $details (@events) { print TEMP "$details";}
close (EVENTS);
```

```
close (TEMP);
rename("/home/aeb/events_temp", "/home/aeb/events_schedule");

$event_details =~ s/^\s+|\s+//g;
print "$event_details\n";
my ($race_time, $race_course, $marketId) = split(/,/, $event_details);


#      Ensure the $race_time format from the schedule is common
#       to the input time from the oddsline

         my ($hours, $minutes) = split(/:/, $race_time);
         if ($hours>12) {$hours = $hours-12};
         $race_time = "$hours.$minutes";

#       loop through oddsline
while (<ODDSLINE>) {

my ($offtime, $course, $horse, $tissue) = split (/,/);
        $offtime =~ s/^\s+|\s+$//g;
        $horse =~ s/^\s+|\s+$//g;
        $tissue =~ s/^\s+|\s+$//g;
        $course =~ s/^\s+|\s+$//g;
if ($offtime =~ /$race_time/ && $course =~ /$race_course/) {
my $horse_bfname = $horse;

#rules for comparing horse names to be implemented here:
#N.B.:  The below *should* work but in practice we need to add more rules, to pick up on
#errors in Betfair horse data, as discussed in part 3, Chapter 13.
$horse_bfname =~ s/'//g;      #removes apostrophes and compares without them
my $horse_bfname_odds = @ {$inter_prices{$horse_bfname} }[0];

#calculate oddsline overround here:
$percent_takeout = 100/$tissue;
$total_overround += $percent_takeout;
push (@horses, $horse);

##      Revalue tissue and ratings for non-runners if necessary:
if ($horse_bfname_odds > 1) { #This means some odds were found - if less than one, the
horse is a non-runner
print REPORT "$offtime, $course, $horse, $tissue, $horse_bfname_odds\n"; #check they were
all captured
$runner = $horse_bfname;
$runners++;
$percent_takeout = 100/$tissue;
$total_takeout += $percent_takeout;
$tissues{$runner} = $tissue;
$bfair_odds{$runner} = $horse_bfname_odds;
# print "$runner\n";
push (@ranking, $runner);
}       #end if horse_bfname_odds

else { print REPORT "$offtime, $course, $horse, $tissue, NON_RUNNER\n";
}
#3 data structures have been created for reuse; @ranking, %tissue and %bfair_odds

}       #end if odds have been found for the horse
};      #end while oddsline

#revalue tissue prices for non-runners,
#store in hash %final_tissues whether non-runners or not

#print REPORT "Tissues = $total_takeout\n";  #if required

if (@horses > @ranking) {
print REPORT "NON_RUNNERS\n";
#revalue here by increasing $tissue price
my $divisor = $total_overround/$total_takeout;
foreach my $horse (@ranking) { my $new_tissue = $tissues{$horse}/$divisor;
$final_tissues{$horse} = $new_tissue;
#print REPORT "$horse, $new_tissue\n"; $new_overround += 100/$new_tissue;
```

```
}  #end foreach @ranking

} #end if #horses exceeds @ranking

else {print REPORT "ALL RUN\n";
%final_tissues = %tissues;
}

#    print "@ranking\n";
#insert alternative selection code here as per section "Extending Decision Criteria"
#for current example oddsline strategy,  divide #runners by 3 and discount the fraction,
#then push to @ranking and loop through it

my $third =  int ($runners/3);

if ($runners > $third) {
@ranking = splice(@ranking, 0, $third);
}

foreach my $contender (@ranking) {
#Get the relevant information to bet...
if ($bfair_odds{$contender} > $final_tissues{$contender}) {

#$bfair_odds = @ {$inter_prices{$contender} }[0];  #more data available if needed
#$bfair_lay = @ {$inter_prices{$contender} }[1];

my $bfair_race = @ {$inter_prices{$contender} }[5];
my $bfair_horse = @ {$inter_prices{$contender} }[4];
my $stake = "5.00"; #or generate automatically, e.g. take the stake from a database

#the following additional parameters are required for bet execution
my $price_asked = $bfair_odds{$contender};    #just making the variable name meaningful
my $betType = "B";
my $betCategoryType = "E";
my $betPersistenceType = "NONE";
my $bspLiability = "0.0";


#print out selections to the betting report if required
#most important is to selections file for subsequent execution
print REPORT "\n$contender, $bfair_race, $bfair_horse, $price_asked, $betType, $stake\n";
print SELECTIONS "$race_time, $race_course, $contender, $bfair_race, $bfair_horse,
$price_asked, $betType, $stake, $betCategoryType, $betPersistenceType, $bspLiability\n";
}       #end if the odds are greater than the tissue
}       #end foreach contender within the top third of the field (i.e. within @ranking)


close ODDSLINE;
close REPORT;
close SELECTIONS;
```

Program Context

For each race, we want to run the program ***bet_formation.pl*** in order to determine selections 5 minutes before the race time. This program presupposes that we have first collected market prices for this race using the program *get_market_prices.pl* and that after the program is finished, any selections for the race will be bet by the program *bet_execution.pl*.

To determine selections for betting, the program brings together inputs from a file holding market prices (***inter_prices***), a file containing the oddsline for all daily races (***formatted_oddsline***), and it writes out a file containing selection details ready for betting, as well as a report on the decisions taken by the program.

In Part 3, we dispense with the interim files and merge this program to run the strategy to collect prices, make a decision and bet within one program. However, the betting decision part of the program is still based entirely on the logic in *Example 9-1*; the programs are more or less concatenated "as is". Thus we can review all the moving parts better by dealing with the decision-making process as a discrete element of the framework, and testing betting decision making alone. In this way, the program or any adapted version of the program can be tested to determine it is producing the correct output based on the received inputs, and only put into production when the bettor is content that everything works and tests as intended

Nonetheless, all programs in the framework can also be run sequentially "as is", relying on the various files produced as inputs and outputs interacting with each other in order to automate the whole strategy. At this stage we assume this is the method of automation, and that our program has been set to run automatically at 5 minutes before the race begins combined with the other programs mentioned, in one executable command, `bet_oddsline_strategy`, as discussed in Chapter 7, *Scheduling – The Key to Automation*. Also, as part of that dynamic scheduling process, we anticipate that a file will already have been automatically generated called ***`events_schedule`*** which will show all events for the strategy to bet in for the day, listed in the order in which they occur.

Our program, ***`bet_formation.pl`***, is set up to read this file to determine which event we are betting on (it will be the first in the list), then to amend the file so that when `bet_oddsline_strategy` is scheduled to run again, the first program to collect prices will read in the next event (i.e. this will become the first event listed) in the file. N.B. The last program `bet_execution.pl` in the current sequence of `bet_oddsline_strategy` has no need to read the `events_schedule` file, so it doesn't matter that the current event information is now changed by `bet_formation.pl`. This is because all information required to bet is supplied in the output file `selections` created by `bet_formation.pl`, as we discuss further in the next section.

Code Walkthrough

Once we have found the relevant event, the objective is to extract and compare the event details, namely the time and course name, with the event details held in the oddsline file, so that we can then find all the contenders with their tissue prices for the race in question, as below:

```
$event_details =~ s/^\s+|\s+//g;
#print "$event_details\n";
my ($race_time, $race_course, $marketId) = split(/,/, $event_details);

#       Ensure the $race_time format from the schedule
#       is common to the input time from the oddsline

         my ($hours, $minutes) = split(/:/, $race_time);
         if ($hours>12) {$hours = $hours-12};
         $race_time = "$hours.$minutes";
```

Dealing with different inputs – such as different data sources with times stored in different formats as above - is a recurring theme once we get to the decision making process. In particular, course names, times of races and horse names that are extracted from different sources need to be standardised in order to compare them. The data can of course be cleaned and standardised at an earlier stage, so that this type of code can be dropped from the decision making program – however, something similar will still have to be written earlier in the day to reconcile Betfair conventions with the data sources used for oddslines, and the resulting values stored in a file or database for later use.

Our program does this for the times above, converting the time format received from ***events_schedule*** to the 12 hour clock which is used to represent the times of races in the file ***formatted_oddsline*** (filehandle **ODDSLINE**), separating hours and minutes by a decimal point as opposed to a colon. This means we are now in a position to compare the times with each other, as below, reading in the ODDSLINE line by line and looking for all lines where the course and time match those of the current event we are assessing for betting purposes:

```
while (<ODDSLINE>) {

($offtime, $course, $horse, $tissue) = split (/,/);

        $offtime =~ s/^\s+|\s+$//g;
        $horse =~ s/^\s+|\s+$//g;
        $tissue =~ s/^\s+|\s+$//g;
        $course =~ s/^\s+|\s+$//g;

if ($offtime =~ /$race_time/ && $course =~ /$race_course/) {
```

Where there is a match on course and time, we want to extract the runner details from the oddsline and apply our betting decision logic to the information associated with those runners. As discussed in the introduction, the first step is to establish which of the declared contenders within the oddsline are running within the race in question (since the oddsline has been put together some time before the runners go to post), and to adjust the tissue prices for non-runners where relevant.

Next, to determine the runner information from the current market, we use the file containing runners and prices for the race that has just been captured from Betfair, ***inter_prices***.

This file was generated by the ***get_market_prices.pl*** program, as shown in Chapter 8. Note that ***bet_formation.pl*** itself makes no calls to the API, emphasising the point that this stage of the framework is all about manipulating existing inputs to implement the rules relating to any particular strategy. In general for production programs, we will typically make the call for prices and then use that information within the context of one program, as we discuss in Part 3, so as to avoid redundancy between programs and to minimise any delay between information changing on the exchange and being used within a betting strategy.

For the purposes of ***bet_formation.pl*** we assume that the file *inter_prices* exists, and in terms of context that this is the second program to be run after ***get_market_prices.pl***. In fact, ***get_market_prices.pl*** and ***bet_formation.pl*** can be set to run sequentially with little difference in practical

delay for the purposes of this specific strategy, so the programs can also be implemented “as is”.

We evaluate the *inter_prices* file below, so that the Perl data structure, a hash of prices called **%inter_prices**, is available to the program just as if it had been retrieved dynamically (note that this code is shown out of sequence within the context of the program, since it is actually executed before the **while** loop):

```
#       read in market information from external file (captured by get_market_prices.pl)
{
open (FILE, '/home/aeb/inter_prices');
local $/;
eval <FILE>;
close FILE;
}
```

This data structure returned is a hash of arrays, so we can look up a number of elements within the array by the hash key (which is a horse name). Before we can look up the price for each runner, however, we must first ensure our oddsline horse name matches the horse name retrieved from Betfair. Of course, a horse has only one name (at the risk of bringing to mind “a horse is a horse, of course, of course”), but it can be represented differently (i.e. inaccurately) by different data sources. Here, we know it is the Betfair convention to remove apostrophes from horse names, so that is what we do, creating another horse name variable by copying our input file’s horse name and removing apostrophes, so that we can then match to any horse name in `%inter_prices` conforming to the transformed name, and also grab the correct price by specifying the correct horse name within the key to the lookup hash, as in the next code snippet.

```
#       note the below does not ensure matching horses without fail

$horse_bfname = $horse;
$horse_bfname =~ s/'//g;

#       To be absolutely sure we should also strip then compare names without whitespace
#       or caps, as per Chapter 13
```

In practice, whilst this should match every horse name according to the convention used, there are occasionally other errors, such as capitalisation of names, within the names loaded to Betfair. We highlight this fact when running a live test on all the framework programs at the start of Part 3. To guard against this we must therefore add in more transformation code in order to match names correctly, as we do with the production version of the code in Part 3. These anomalies, although rare, are also dealt with in the commented lines above.

Next, the code moves on to the job of identifying any non-runners – using the matched horse Betfair names, as discussed above - and making a simple adjustment in the tissue price as a result of that. Based on the new runner information we could also recalculate all tissues from the ground up, using the original input data, according to the method used to generate the oddsline in the first place. The method in the example program is more generic, if blunter, in that we readjust prices based upon the first tissue calculated.

First we calculate the percentage takeout of each runner in the book using the tissue for all the declared runners, and save the **$total_overround** in a separate variable (this should already be at, or very near to, 100%, of course).

```
my $horse_bfname_odds = @ {$inter_prices{$horse_bfname} }[0];

#calculate oddsline overround here:
$percent_takeout = 100/$tissue;
$total_overround += $percent_takeout;
```

In order that all the contenders for the particular race we want to assess are available later, we save the horse names for this race to a discrete array also.

```
push (@horses, $horse);
```

Next step is to check for non-runners, if any, in the race. We do this by seeing if a back price exists for each Betfair horse name. If a price does not exist (i.e. `$horse_bfname_odds` is not greater than 1), then we assume that the horse is a non-runner.

```
##      Revalue tissue and ratings for non-runners if necessary:

#This means some odds were found - if less than one, the horse is a non-runner
if ($horse_bfname_odds > 1)     {

#check they were all captured
print REPORT "$offtime, $course, $horse, $tissue, $horse_bfname_odds\n";
```

Provided we have made all attempts to match to the correct horse name, a simultaneous test for the presence of a back price is a useful short cut to determine non-runners for any British or Irish horseracing market 5 minutes prior to the off. Even from the earliest phases in such markets (i.e. when they are made available the day before racing) there is a back amount, even if only 1.01, although lay amounts – i.e. amounts on the exchange asking for a layer – may frequently be missing. By the same token, it should be employed carefully on sporting events with no liquidity, e.g. some US racing. In such events, perhaps where the bettor wants to be one of the earliest layers, a call to **GetMarkets** first, in order to explicitly extract and then match non-runners (since this is also a method of retrieving non-runner information) is preferred.

Having captured the horse details, we print to the filehandle **`REPORT`**, which is our betting log, a row of information for the horse details captured, including the tissue price from the oddsline for each horse captured and the Betfair odds for the horse.

Next, we create a new array **`@ranking`** corresponding to all the Betfair horse names we have, those being the runners in the race (we already have one array **`@horses`** corresponding to our oddsline of declared runners); we again will calculate the percent takeout of each runner and the total takeout, or overround, for the remaining runners, as we did for the declared runners.

```
$runner = $horse_bfname; $runners++;            #
$percent_takeout = 100/$tissue;
$total_takeout += $percent_takeout;
$tissues{$runner} = $tissue;
$bfair_odds{$runner} = $horse_bfname_odds;
#                                       print "$runner\n";
push (@ranking, $runner);
}       #end if horse_bfname_odds
```

Note that **`@ranking`** above is an array which has all our actual contenders ranked in order of best chance to least (since that is the way that we wrote the input file). We therefore use a slice of this array in order to extract the top third runners who are ranked the highest.

At this point we are done with processing runners in the race, provided that there are no non-runners, and are ready to proceed to selecting those horses we want to back as part of this strategy. However, if there are non-runners found according to our criteria, we print those out to the program report, including the same oddsline details as for a runner, appending NON_RUNNER in place of the price information, so this can easily be reviewed.

```
else {
print REPORT "$offtime, $course, $horse, $tissue, NON_RUNNER\n";

}
```

Output from the report can be seen in the first Chapter of Part 3, when this program is used in sequence with the other examples in the framework to run this strategy on a live basis.

Having set up data structures to deal with the event of non-runners (and the adjustments needed to the oddsline as a result) we now see if there are any. The acid test is to compare the size of the array **@horses** against **@ranking**. If the declared horses in **@horses** are more than the actual runners "going to post" in **@ranking**, we know that there are non-runners, and therefore must revalue all the tissue prices from our oddsline, creating new tissue prices. If that is the case, the code in the next block does this simply by calculating the percentage difference in the overround between the oddsline and actual prices (as represented in the code below by the variable **$divisor**).

```
#3 data structures have been created for reuse; @ranking, %tissue and %bfair_odds

}
};

#revalue tissue prices for non-runners, store in hash %final_tissues whether non-
#runners or not

if (@horses > @ranking) {
print REPORT "NON_RUNNERS\n";
#revalue here by increasing $tissue price
$divisor = $total_overround/$total_takeout;
foreach $horse (@ranking) {
$new_tissue = $tissues{$horse}/$divisor;
$final_tissues{$horse} = $new_tissue;
#print REPORT "$horse, $new_tissue\n";
#$new_overround += 100/$new_tissue;
}
```

The tissue price is adjusted by **$divisor** which has the effect of lowering the sum of tissue prices for each contender by the sum of probabilities that have been removed from the book. Of course, each individual contender's odds will also be reduced – these are calculated and saved to a new hash, **%final tissues**. We also state again in the report summary that non-runners have been found in the race, printing NON_RUNNERS to the betting report.

The second part of the **if** statement deals explicitly with the situation where all go to post. Here we save our hash of tissue prices with the same name as the tissue hash for non-runners, so we can use the same selection process later in the program. That is, we can refer to the final_tissues hash of tissue prices for each name, and that it will just work.

```
} else {print REPORT "ALL RUN\n";
%final_tissues = %tissues;
}

#               print "@ranking\n";
```

Finally we are ready to make selections by applying the strategy to consider any overlay opportunities in the top third of the field.

First, we work out what a third of the (remaining) field is, saving the integer result to a variable by which to peel out the first third of runners in the next step. This means we will consider the top 2 horses in a field of 8, the top 3 horses in a field of 11, and so on. Alternatively it is clear we could consider any number of contenders at this point, chosen in many different ways - this is simply the rule for the example strategy.

```
$third =  int ($runners/3);
```

Next, we take an array slice from **`@ranking,`** effectively reducing **`@ranking`** only to the top third of the field. (N.B.. By stating that the number of runners must be greater than the variable $third, we are effectively eliminating 2 horse races, since there have to be at least 3 runners for $third to be an integer and to implement a strategy that considers the top third of field. If **`$third`** is not an integer of at least 1, the **`@ranking`** array will be spliced by nothing, so nothing will be left in it. If wanting to consider the top horse in a two horse race we could instead say `if($runners > 3)`, meaning that @ranking would be unchanged for any runners which did not fulfil this condition.)

```
if ($runners > third) {
@ranking = splice(@ranking, 0, $third);
}
```

Finally we are ready to compare the back price odds from Betfair with the final tissue prices (readjusted where necessary) from any oddsline.

Below we do that for each contender in the candidates for betting left in **`@ranking`** by looking up the relevant prices from the hashes we have created earlier. If the Betfair price is greater than the oddsline, we get the relevant details for that horse and write it to a new file, ready to be picked up by a betting execution program, as covered in the next chapter, Betting Execution. The details are all those we need for the **PlaceBets** call, which we use in the next Chapter, including the Betfair identification numbers for **marketId**, **runnerId**, and **stake**. We also write the same details to our program report.

```
foreach my $contender (@ranking) {
#Get the relevant information to bet...
if ($bfair_odds{$contender} > $final_tissues{$contender}) {

#$bfair_odds = @ {$inter_prices{$contender} }[0];  #more data available if needed
#$bfair_lay = @ {$inter_prices{$contender} }[1];

my $bfair_race = @ {$inter_prices{$contender} }[5];
my $bfair_horse = @ {$inter_prices{$contender} }[4];
my $stake = "5.00"; #or generate automatically, e.g. take the stake from a database

#the following additional parameters are required for bet execution
my $price_asked = $bfair_odds{$contender};    #just making the variable name meaningful
my $betType = "B";
my $betCategoryType = "E";
my $betPersistenceType = "NONE";
my $bspLiability = "0.0";
```

```
#print out selections to the betting report if required
#most important is to selections file for subsequent execution
print REPORT "\n$contender, $bfair_race, $bfair_horse, $price_asked, $betType, $stake\n";
print SELECTIONS "$race_time, $race_course, $contender, $bfair_race, $bfair_horse,
$price_asked, $betType, $stake, $betCategoryType, $betPersistenceType, $bspLiability\n";
}       #end if the odds are greater than the tissue
}       #end foreach contender within the top third of the field (i.e. within @ranking)


close ODDSLINE;
close REPORT;
close SELECTIONS;
```

That's it for this example; we have produced a file with selections ready for betting by the next program in the sequence, and a report stating what those selections are.

Extending Decision Criteria

There are any number of input variables to consider which we can use to dramatically change the nature of the betting decision to be taken, and hence the strategy. However, any betting decision will have in common the need to stipulate the number of selections, the price required, and stake for each.

To give a flavour of how we can start to adapt existing code to transform the original example bit by bit to incorporate these elements, (until at a certain point it actually looks like a different strategy altogether), let's consider changing only the final block of code in *Example 9-1* (at the point where we are determining the total number of selections to bet in the oddsline, the price that we wish to back each selection at, and the stake we wish to use).

Number of selections and minimum price

Using *Example 9-1* as a base, we can keep 90% of the code but substantially alter the decision criteria and betting strategy for any number of parameters by slightly altering the final block of code.

For example, the strategy shown implies considering and usually backing more than one horse in any race, and to variable total stakes per race, despite fixed unit stakes per runner. Therefore, bigger fields have the potential to create far more bets.

Changing the criteria – for example to consider the top contender alone – might bring a consistent number of bets per race (a maximum of one per race, in this case) combined with a consistent maximum stake for every race. We can keep the oddsline intact as the input file, but change the strategy to consider only the strongest selection made by the underlying data, in the form of the highest rated contender

To implement this in the final block of code for *Example 9-1*, we can remove the foreach loop:

```
foreach $contender (@ranking) {
```

and simply modify the next line consider whether or not the top contender as represented by the variable **`$ranking[0]`** is an overlay, as in:

```
if ($bfair_odds{$ranking[0]} > $final_tissues{$ranking[0]})
{
```

If it's an overlay we write it out as a selection, if it's not we leave it. Limiting the selection to one potential bet per race makes for more straightforward backtesting of our estimated odds over time. We compare the expected number of wins predicted by the oddsline top selection with the number of times the top rated selection actually wins, then see if the proportions tally, and therefore whether there is indeed an edge in the oddsline over the market. Refining the oddsline in this way is discussed further in Part 3.

Finally, let's say that the top rated selection was not produced by an oddsline but a system method. For example, one which had been shown to be profitable through backtesting results for horses conforming to certain criteria, with an added price condition - such as "starting prices must always be greater than odds on".

Provided we have applied the rules of the system to an appropriate data source in order to generate a file of selections (in place of the oddsline), the program can be adapted to process any set of contenders with the appropriate rules to back them.

In this case, the line from the input file containing the selection would also specify the race time and course, so it can be picked up at the relevant event time, as well as stipulating the minimum price for the systems method (this would replace the *`formatted_oddsline`* file imported at the beginning of *Example 9-1*). Implementation of the system rules are thus to capture selection details (much as we do for our existing oddsline example), and then apply a simple minimum price condition – thus the condition that the price for the top ranked selection is greater than evens (for example), in order to constitute a bet:

```
if ($bfair_odds{$ranking[0]} > 2]})
```

Dutching

In the strategy implemented in *Example 9-1*, we are always (potentially) betting more than one contender, depending on how many overlays are found for the oddsline versus the market, which in turn can increase dependent upon the field size (given we always consider a fixed proportion of the field). As such, the standard example is dutching to a level unit stake, and variability in overall outlay per race is high. We can, however, implement a dutching method in many other ways including:

- Level stakes for a varying number of horses
- Targeting a maximum outlay per race, with staking divided amongst the resulting contenders
- Targeting outlay to ensure the same profit if any of the overlays wins.

To implement such a dutching strategy for the example oddsline, we can again adapt the final block of code in *Example 9-1* by closing the block before we write out staking information, in order to first determine how many contenders are final bets (or selections). To do this, we create another array **`@selections`**, to contain the name of each contender. Then, we act upon this array to apply dutching rules and determine final stakes according to one of the criteria above:

```
foreach $contender (@ranking) {
if ($bfair_odds{$contender} > $final_tissues{$contender})

#for dutching for the same profit, work out sum of odds first
$bfair_odds = $bfair_odds{$contender};
$total_odds += bfair_odds ;
{
push @selections, $contender;}

foreach $contender (@selections)  {

#Determine the stake according to the dutching strategy, as described in text below
#Calculate stake here, according to method
#eg:
#$max_outlay = 30;
#$number_selections = @selections;
#$stake = $max_outlay/$number_selections

$bfair_race = @ {$inter_prices{$contender} }[5];
$bfair_horse = @ {$inter_prices{$contender} }[4];

#the following additional parameters are required for bet execution
my $price_asked = $bfair_odds{$contender};    #just making the variable name meaningful
my $betType = "B";
my $betCategoryType = "E";
my $betPersistenceType = "NONE";
my $bspLiability = "0.0";

print REPORT "$contender, $bfair_race, $bfair_horse, $price_asked, $betType, $stake\n";
print SELECTIONS "$race_time, $race_course, $contender, $bfair_race, $bfair_horse,
$price_asked, $betType, $stake, $betCategoryType, $betPersistenceType, $bspLiability\n";

}
}
close ODDSLINE;
close REPORT;
close SELECTIONS;
```

To stake the same amount for each runner, we can stipulate our maximum outlay as a variable, let's say £30 per race

```
$max_outlay = 30;
```

then divide by the number of selections

```
$number_selections = @selections;
$stake = $max_outlay/$number_selections
```

So that we have the same outlay on every race, but a variable outcome, depending on how many runners to divide the stake over and what horse wins at what price.

We could also stipulate a desired return for each contender, in the same fashion:

```
$desired_return = 100;

#within context of loop
```

```
$stake = $desired_return/$bfair_odds{$contender};
```

This method of dutching can be useful if dutching the field in the event of an overbroke book, e.g. For a market at 98%, where we know by successfully backing all runners using the method shown we will guarantee a 2% profit on our **$desired_return**.

In the case of executing our oddsline strategy on a number of races however, stipulating a desired return would mean our total stake is still variable depending on how many runners we want to dutch.

To dutch bets in order ensure the same profit no matter which of our oddsline selections wins (assuming of course that one of them wins), and to ensure that our stake is fixed for every race, we can use a different method again. We still work off our **$max_outlay** variable but in the first instance need to calculate what proportion of that outlay we need to stake for each contender.

Here, we simply need to sum the total decimal odds of all selections and retain that as a variable, on our first loop through the code to determine overlays, as in:

```
$bfair_odds = $bfair_odds{$contender};
$total_odds += bfair_odds ;
```

Subsequently, when going through the selections loop for each contender, we determine stake proportion allocated to that contender as follows:

```
$stake_proportion = $bfair_odds{$contender}/ $total_odds;
```

Finally, the stake is calculated by taking the relevant proportion of **$max_outlay**, and formatting the stake value to two decimal places (so it will be accepted in the **PlaceBets** call during betting execution, a process covered in Chapter 10).

```
$stake = $stake_proportion * $max_outlay;
$stake = sprintf("%.2f", $stake);
```

Changing the price dynamically

The currently available back price was taken as the price at which we want to "get on" in *Example 9-1*. What if we want to put in a price for each selection above that, in an attempt to beat what the market is currently prepared to offer?

Of course, attempting to do so will grant no guarantees that we will be matched at a higher price, and indeed, unless there is a very good reason to suspect otherwise, there is every chance we could end up with a worse price - being left "unmatched" and still faced with the problem of getting on for the strategy. Creating unmatched bets also adds complexity to any automated implementation in that we will need to track them after the betting execution stage, and perhaps update them if the price goes against us and we still want to "get on".

However, bearing in mind those caveats there are circumstances where this may make sense. For example, if a predictive mechanism for prices has also been implemented (and works, which is another matter) and the automatic bettor is confident such a price will be reached.

Or we may surmise that it if we are betting early it is worth putting in speculative prices on the basis that they may be matched due simply to the increased chance of volatility between execution of the bet and the event off time.

A practical example for changing the price in an attempt to beat the market (which does not require complex prediction based upon price history) is attempting to be first in the queue at the next market price where the market spread (the spread being the difference between the back and lay price) is more than one tick size. For example. if the back price is 10 and the lay price is 11, we would prefer to be first in the queue to back at the next Betfair tick value, which is 10.5, as opposed to taking the freely available 10.0.

The acid test is if the bettor is convinced they have a method which produces better prices over time than settling for the currently available back price alone. Such a method can of course be researched and tested prior to implementation.

There are a number of ways of automatically specifying what that better price should be. Staying with the spirit of making small changes to the final block of code in *Example 9-1* in order to produce some quite dramatic differences in results, we can simply specify that a back bet should be placed at the current lay price rather than the back price before writing out our bets to file, i.e. `$back_price = $lay_price`

This will have the effect of incrementing the price by the size of the current spread.

For a more generic solution to extend options for stipulating bid and offer prices in relation to current prices, we can implement a simple subroutine to increment or decrement prices by specific tick sizes. This subroutine can easily be adapted to accommodate other requirements for changing the price dynamically in betting strategies, such as increasing the price by percentage amounts.

It is worth pointing out again that the reason for implementing a subroutine to take care of this is that any price must correspond to the exact format of the Betfair price increment, otherwise bet execution will fail. If we are taking the current back price returned by a function to capture market prices this is not an issue, since the format is already compliant. However, to change the price it is not enough to add or subtract from this price. Specifically, the Betfair price must correspond to one of the 350 incremental prices – from 1.01 to 1000 - in the Betfair price array. Adding 0.1 to a back price of 8.4 will result in a rejected bet (since 8.5 is not a valid Betfair price) so we have to specify “+1 tick”, for the next valid increment, or a price of 8.6.

We can think of this array as a ladder of prices. In order to automatically move up and down that ladder of prices, we simple need to create as an array (in Perl terms, or whatever other programming language is being used) all the Betfair prices available. The incremental table for Betfair prices is shown in Appendix 5; prices must be entered in the exact format shown.

We must make this array present to the program, and a number of possibilities are available once it is. For example, let’s specify a subroutine to increment a currently available back price by a number of ticks, as discussed above.

Example 9-2: Subroutine to increment back price dynamically

```
#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#


sub step_odds {

my ($price_to_increment, $step_size) = @_;
$price_to_increment =~ s/^\s+|\s+$//g;
$step_size =~ s/^\s+|\s+$//g;
my $count = 0;
my $array_position = ();
foreach my $odd (@Betfair_odds) {
if ($odd == $price_to_increment) {$array_position = $count};
$count++;
}


my $ladder_value = $array_position+$step_size;
if ($ladder_value <0) {$ladder_value = 0};
if ($ladder_value > $#Betfair_odds) {$ladder_value = $#Betfair_odds};
my $stepped_odds = $Betfair_odds[$ladder_value];
return $stepped_odds;

}
```

The above subroutine loops through the array of Betfair prices in order to find the array index of the first argument (i.e. the price to increment) and then adds or subtracts the second argument (depending on whether we wish to move up or down the price ladder), from that index position.

Since we are working with the array index position, as opposed to the array of prices itself, we are working in terms of the 350 ticks available, not adding or subtracting prices themselves. Using our supplied arguments gives us the array index of the required price, so we can return the value (i.e. price) associated with it, as follows:

```
my $stepped_odds = $Betfair_odds[$array_index]).
```

The subroutine is written to take arguments for the price to be modified (it could be a back or a lay price), and the number of Betfair ticks by which to modify the price.

How is this implemented in practice? Let's say we have good reason to believe the current back price of 15.5 may drift by a further 2 ticks prior to the off, or simply want to speculate on matching at a higher price:

```
$price_to_increment = $back_price;
$ticks = 2;
$new_back_price =  step_odds ($price_to_increment, $ticks);
```

Remembering that **`step_odds`** is the name of the subroutine, it gives us a **`$new_back_price`** with a value two ticks higher than our current back price of 15.5, so 16.5, which we can write out to our selections file ready to bet, if used within the context of *Example 9-1*.

Such a subroutine can equally be implemented at the betting execution stage, or used within the context of one program to fetch prices, make decisions and bet.

Using the Market Overround for Optimal Pricing

Without changing the price acceptance criteria on our example program we can use the market overround to determine whether or not the price being offered is “a good deal”. If, in the case of our example, we are pursuing a backing strategy, we will only decide to accept back prices that are available within the context of a minimum market overround, as close to 100% (and in an ideal world less) as possible Whilst the market overround will not guarantee the best price available on any particular selection, it will over the long run guarantee that the best prices within the market are consistently obtained for a given strategy.

To obtain a price conditional on a market overround we need to wrap any decision making process that writes out bets within a loop that requires the minimum overround is attained first.

The level of the overround is up to the bettor. It can be used as a constraint – under 100% would ensure most bets were unmatched – or as a “sanity check”; for example, less than 103% is usually achievable in most UK horseracing markets near the off, meaning we are likely to be matched but will still avoid taking a price within a poor market or, in the worst case, when a market anomaly occurs. Market anomalies would be, for instance, when a runner is withdrawn and the market reforms, temporarily creating a large overround. This is a problem for an overlay strategy, since the price may still be determined as a value bet if the program visits the market during this transition time, even if the price has temporarily halved and will soon return to near its previous level.

Obtaining the market overround was discussed in Chapter 8, *Capturing and Using Market Prices*. We also showed a loop through process of obtaining prices with calls to the **GetMarkets** and **GetMarketPricesCompressed** services in that chapter. By amalgamating the betting decision making process with the capture of market prices, we can enclose the whole within a loop that only moves forward to betting execution if a certain overround is reached. We show examples of this in Part 3.

Price Prediction and Unmatched Bet Considerations

Putting in offers that are not accepted will add complexity to our implementation, but, as we discussed in the section on changing prices, it can be worthwhile depending on whether the methods used mean that the bet is matched at better prices in the long run.

Within the context of an overlay strategy, there is a further dilemma over whether obtaining a better price is speculative or conditional to the betting strategy. For example, a contender may be deemed an overlay at 10.5 to back, yet the price is currently 10. In such cases it seems prudent to put in an offer at 10.5 (provided the rules of the strategy require an overlay only, with no percentage buffer). However, what to do if the current price is 8? Should offers be put in on all bets?

If the motivation behind the decision to obtain a better price is speculative then the payoff has to be weighed against the downside (of not being matched) and additional implementation considerations. We can define a speculative decision to do this as the automated equivalent of "chancing our arm" on obtaining a better price (in other words, if we have already determined that the bet is value but want to see if we can get a better price). By contrast, to determine whether or not it is worthwhile to pursue a conditional price to fulfil the betting strategy, we might wish to build some logic into the program that assessed the prices available to determine:

a) If the market was close to the price, putting an offer in a price to the nearest tick that fulfilled the value criteria
b) Putting together some logical criteria to determine the likelihood of such a bet being matched. If the spread between bid and offer price is 2 ticks and the nearest tick (which we want to occupy) has no money available, that could qualify.

So with those caveats defined, useful considerations to determine whether to go down the route of putting in a bet outside the current market (i.e. outside the current bid and offer prices) are:

- Whether it is conditional or speculative to the strategy
- Within this context, assessing the likelihood that the price and volume we are asking for will be matched

Within the context of "how likely the price is to be matched", of course we cannot "know" whether we can achieve a price or not but we can attempt to make a prediction. Parameters to consider within such a prediction are:

- How far the current market price is from the required price
- How long the market has to run – in trading terms – until the event starts.
- How informative market price and liquidity information can be

Indicators which can help with such decisions are discussed in Chapter 8, *Capturing and Using Market Information*. The last section of this chapter discusses the overlap with trading strategies in using such indicators.

The Betfair SP Market and Bet Persistence

In the example strategy we specify use of the exchange when writing the bets out to a file, therefore ensuring that the bets expire, if unmatched, at the time of the off. Other options, in using the Betfair SP Market and Bet Persistence (still using the exchange but letting a bet ride unmatched into the "in-running" market), are also available to our strategy. We explore these options briefly below and cover implementation and parameters for these options in more detail in Chapter 10, *Betting Execution*.

The **Betfair SP Markets** offer the prospect of automatic matching but at an unknown price. However, we can stipulate a minimum price for whether or not a bet will be placed at SP. In this sense the SP market with a minimum price requirement is similar to specifying an unmatched bet at a conditional exchange price. It poses the same

issue for the automatic bettor executing a strategy prior to the off as whether or not to take the currently available back price or wait for a better exchange price.

As far as the automatic bettor is concerned in trying to back or lay the result of an outcome, any offer outside the currently available price, where we know we will be matched, is a case of swapping certainty for uncertainty, with no guarantee of a preferable result.

On the other hand, the advantage of the SP market is that there is guaranteed liquidity, and some certainty in "getting on". Therefore, we can pursue a strategy to see if a bet can first be matched within the exchange market, asking for a better price than is currently available, or switch the bet to the currently available best price, or the SP market prior to the off. This can be an option to ensure that bets are matched, but still leaves uncertainty on the price. To be sure of executing the example strategy, we thus take the available back price.

The **Bet Persistence Markets** enable the bettor to maintain bets placed on the exchange market once those markets have turned to "in-play". These markets provide an interesting extension to automated strategies, which also offer the possibility of creating a further trading or hedging strategy. For example, an objective in this case could be to use the bet persistence feature for an automated trade which predicts price behaviour based on the fundamentals of a horse's in-running characteristics as opposed to studying price movements. For example, we can identify likely front runners in a race based on fundamental data analysis, and back them in the exchange markets prior to the off (ensuring the bet is placed), at the same time placing a lay bet at a much lower price with the condition of bet persistence. This would be a "back to lay" strategy, with the intention that the lay price becomes available during the in running markets (on the basis that any front runner's price will contract once "in-play"), thus providing a profitable hedging or trading opportunity. Of course, such a strategy requires its own research into what reasonable price drops can be expected and insofar as what risk the bettor is prepared to accept that the bet is unmatched.

However, for a strategy concerned simply with obtaining best price on an outcome prior to the race, using Bet Persistence is rather a "shot in the dark" in order to obtain a better price without any reference to the horse's running style. There is of course the possibility of a match at a higher asking price, provided the runner lengthens in-play. By the same token there is a good chance that the runner will lengthen in price because it is clearly going to lose, as opposed to getting on at the required price when it still has every chance. To cap it all, there is no guarantee, even if the horse wins, that the in-running market will produce a better price than that available on the exchange before the off.

Thus bet persistence may offer little advantage and high risk for obtaining optimal price on a single bet - unless there is very good reason to think that the contender may lengthen or shorten in price due to its running style, or due other factors related to the way the race will develop in-running. At this point, the logic is less geared towards obtaining best price alone for our first strategy betting on a particular conclusion to the race, but on a combination of separate predictions. In such cases, a better opportunity for using Bet Persistence can exist in *lay to back* or *back to lay* strategies. Such a strategy can be used to provide a hedging opportunity (if leaving some of the stake on the outcome) or used to profit on trading movements in prices alone. This type of

trading strategy differs from the norm in that the underlying reasons for the trade rest on using fundamental data to predict how the race will be run, as opposed to using "anonymous" price data obtained from the market alone.

Automating staking plans using a betting bank

Example 9-1 uses a fixed level stake in order to back each selection from the strategy. The dutching strategies analysed in the previous sections would further divvy our stakes on a per race basis, changing the strategy to some extent and making stakes proportional to the dutching method, with an overall limit suggested per race. However, whether we choose to bet on a per race or a per bet basis, there is ultimately a wider logic to be applied to the question of staking in terms of how to determine what overall stake to put at risk within any given risk. This does not form part of our example strategy, but any production strategy should consider the question of automation with regard to staking plans.

An ideal staking plan seeks to avoid the risk of ruin and maximize returns. At a basic level any staking plan is related to the concept of using a betting bank.

As discussed in Part 1, staking plans have always been fertile ground for debate with regard to the difference they can make to profitability of any given strategy. The general consensus, and a consensus which it is as well to heed, is that no staking plan can turn a strategy which does not show a profit at level stakes into a profitable one. Of course, if a sequence of winning and losing bets is known in advance, it is possible to construct such a plan, but this is backfitting *par excellence*. Alternatively, with theoretically unlimited funds and unlimited liquidity in markets (such as the infamous Martingale-based strategies), it is possible to increase stakes to cover losing runs, but in the real world with practical limitations on market liquidity and funds this can be a rapid path to ruin.

More accepted staking plans to maximize returns rely on quantifying the advantage which a selection method has over the actual probability of an event occurring. The most widely touted of these is the Kelly Criterion, discussed in Part 1, Chapter 1, in the subsection *Staking,* which concentrates on the advantage which each individual bet has over the market, and aims to stake a higher proportion of the betting bank whenever that advantage is greatest.

Since the example strategy is itself based on an estimated probability for each contender, resulting in a tissue price, the full range of staking plans, including Kelly, is potentially available to us within our example strategy. However, it needs to be stressed that the level of confidence in the edge remains a critical factor to test and validate if pursuing Kelly, and so far in our example this has not been established. So we have an input to calculate Kelly, but it is possibly unreliable. To implement a full Kelly strategy the advantage would have to be absolutely tested and reliable; otherwise, the method may still bring advantage over level stakes, but needs to be diluted in accordance with the level of confidence in the testing of the oddsline; hence moderated Kelly strategies, such as "half Kelly", "quarter Kelly" and so on.

The important elements to consider from an implementation point of view are:

- Dynamic calculation of betting bank
- Definition of perceived edge
- Algorithm for calculating stake

Other widely used staking plans (in addition to level stakes) are betting a percentage of the betting bank to level stakes with each bet.

As with record keeping, the manual process of recalculating stakes associated with a betting strategy (even if a simple staking plan, let alone calculating Kelly for multiple contenders prior to each race) is a tedious one, which can be resolved simply and as often as we like within the context of an automated program.

We can set a program to calculate stakes as a proportion of the betting bank after every bet, day, week, month, season, or year. We can adjust stakes based upon probability or "edge" of each bet, as with Kelly, implement stop loss rules, and so on.

Any staking algorithm can generally be implemented programmatically (a good discussion on implementing Kelly is covered in John Haigh's *Taking Chances* in the Bibliography). The definition of perceived edge – and also confidence level in that edge - is down to testing of the strategy, and can be supplied as an input. The final piece of the jigsaw, if wishing to automate the whole process, is that we need to have a betting bank balance available as an input.

A betting bank can be maintained automatically in a database table that is updated with results of each bet, and keeps a running total – or, more simply, it can be a betting exchange account balance, or a combination of the two (e.g. we may not wish to tie all capital in a betting exchange account, preferring instead to take the outstanding balance and combine it with a residual bank balance that does not sit on the exchange, or with other accounts that reside elsewhere). In terms of using a betting bank for staking plans, the betting bank itself need not constitute all available betting funds, but those available for the purpose of any particular strategy, and thus setting up one betting bank for a strategy applied to one account is often an expedient solution.

As far as using functions from the Betfair API to help with this process is concerned, calls such as **GetAccountFunds** will provide a balance as an input for an automated staking plan to work with. If we wish to split the betting bank balance according to those bets placed by strategy (and use different staking plans per strategy, but within one account), we can also use **GetAccountStatement** for a detailed betting history. We cover the use of **GetAccountStatement** for the purpose of record keeping, split by betting strategy and other parameters, in Chapter 11.

In the meantime, to show how we can implement dynamic staking within the context of the framework, let's choose a simple staking example which uses account funds and therefore assumes one account is used for the strategy. We therefore assume that available funds in an account reflect the betting bank, and in terms of the staking plan, that we want to stake a fixed percentage of the betting bank as the unit stake for each bet.

The context of the decision making program is a great place to insert a block or a few lines of code to implement such a plan. Therefore, going back to our original *Example*

9-1, to see how we can incorporate dynamic staking into the context of the decision making program, we can see that it is easy to alter one line of code to add such functionality. We could use the following (instead of `$stake = £5` to stipulate a fixed level unit stake) to calculate the stake dynamically:

```
#stake = available betting bank/N OR algorithm to determine proportion of bank to use

my $funds = get_account_funds($token)
my $availBalance = $funds{availBalance};
my stake  = sprintf ("%.2f", $availBalance/200);
```

To test this functionality, we show a program which only uses the above lines to demonstrate the **`get_account_funds`** function combined with a staking plan in *Example 9-3*.

Example 9-3: Automatically determining stakes using an account balance

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#

#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";           #Free Access API access code

#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }

#stake = available betting bank/N OR algorithm to determine proportion of bank to use

my %funds = get_account_funds($token);
my $availBalance = $funds{availBalance};
print "Available betting balance is:  £$availBalance\n";
my $stake  = sprintf ("%.2f", $availBalance/200);
print "Stake for the next bet (based on staking 0.5% of betting bank) is:  £$stake\n";
```

Note in *Example 9-3* we use "available balance" from the account as opposed to "balance" in order to play on the safe side of the betting bank (e.g. it may be that bets which have won have not yet been settled, in which case they would still be liabilities on the overall bank, thereby reducing the available balance and limiting stakes – we instead assume that all liabilities are just that).

For Kelly and other plans, the program logic remains the same (get account funds, then ; apply a staking algorithm to the balance based on other inputs); we simply need a more complex algorithm, and to rely on the probabilities assigned by our oddsline.

Further use of automatic account functions is also explored in Chapter 11, *Automating Record Keeping*.

Automating trading decisions

We have generally made a distinction thus far between betting and trading. The distinction is that the focus of automatic betting is to profit from outcomes of events rather than price changes in markets; due to differences in objectives, we concentrate on betting and consider elements of automated trading when it crosses over.

However, as pointed out in the introduction, there is clear overlap between the tools and mechanisms used for both trading and betting strategies within the context of capturing and using market prices, then making decisions about placing bets at this stage of the framework.

Therefore, a brief summary on some of the differences in implementation between betting and trading strategies, plus how the reader might use the framework to develop trading strategies is appropriate at this stage.

In terms of using API services programmatically, there is of course no difference in the raw services that are used, such that the atomic elements available for trading are already available. Building upon some of the derived functions for market overround, price momentum, average traded price, and so on will develop an armoury of tools needed by the trader at the market information stage. However, these need to be combined with other stages of the framework, also adapted for pure trading.

The following stages in the framework can be taken as a "mini-framework" for trading in their own right.

- Capture of market prices
- Decision making process
- Betting execution

We cover elements of the betting process that can be adapted for trading within each of the chapters relating to the above stages of the framework.

In terms of programming effort, the emphasis shifts to where the bulk of the implementation is concentrated – in trading, the process is entirely about predicting (or profiting on the volatility of) the market itself; in betting on outcomes the process starts

with a prediction or an assessment of the probability of the outcome, and goes to the exchange to ensure maximum profit on that assessment of probability.

It's analogous to different objectives in financial markets, where algorithmic trading can be used for execution only, to break up orders on electronic exchanges for achieving the best price on financial instruments (where there has been a decision to invest based on fundamental analysis), or can be used entirely as a vehicle for predicting and profiting on movements in tick prices.

The critical difference in the complete automation of trading strategies is generally that there is a need to visit the market multiple times, to take and then close a position - and thus to loop through the "mini-framework" multiple times.

Simple trading strategies – for example placing a simultaneous back and lay bet in the right circumstances, or delayed by some margin determined by an automated analysis of prices – are simple to implement. On the return leg of the trade we might "green out" or take a loss on the position, or wait until a subsequent visit As we add more loops to emulate the decision making process that an exchange trader might use, however, the implementation becomes more complex.

It should also be remembered that the preferred option for many betting exchange traders is to use automated utilities that provide shortcuts to tasks that are impossible manually, such as real time price indicators (average prices, momentum, and so on) combined with utilities that enable efficient interactive execution such as one-click trading, viewing the entire price ladder for each contender, viewing multiple market favourites (since these are the candidates for highest liquidity and trading potential), and so on. Many packaged products provide such utilities based upon the API. All such utilities can be created or tailored to the individual within the context of API services discussed (with the addition of a GUI). However, the critical point is that the overall control mechanism remains with the trader in interactive mode, so that any risk on positions retains human control.

In other words, all of these options can be programmed for complete automation. The question is rather down to the trader's mindset and an examination of their own strategy, to determine how much of a successful approach is currently done by "feel" (or even luck), responding to the presence of multiple indicators, and taking quick decisions. Or, how much of a strategy is down to a significant indicator (or combination of indicators) that can be programmatically spotted and traded for profit. No doubt some nuance and flexibility will be lost in automating a series of complex trading interactions. Keeping it simple is therefore a good approach – complexity can always be added if the fundamental strategy is sound.

Chapter 10: Betting Execution

There has been a lot of preparation up to this stage – with about 50% of the effort on implementing the method and rules of the strategy and 50% on complete automation of that process. At last, we are finally in a position where our automated betting strategy or "robot" is ready to execute bets - be it to back or to lay, put in a price at the market or a speculative offer. In short, we are ready to execute on the decisions made during the last phase of the framework, and translate them to an actual bet or series of bets on the exchange. Within this stage of the framework, we also include monitoring and updating bets after placing them, since this is all part of ensuring effective execution.

In terms of the traditional betting process (still available at a licensed betting office near you!), we are engaged in the automated equivalent of scribbling out a betting slip, shielded from prying eyes (but no longer in a smoky corner) on which is transcribed the stake, name of the horse, meeting and time of race. We are about to visit the counter. This would be true of a backing strategy; of course, if laying, we are in the more unusual position of sitting in the manager's office totting up our liabilities on the slips that are coming in.

Similarly, in the automated process, we have all the details needed to execute the bet – or a few bets - stored now in a file on a computer disk, instead of a betting slip. (Later in Part 3, when we join stages of the framework to create single programs embodying the whole framework, we will not bother writing the bets to file, but rather store and reuse the details in the context of a program using available memory).

Moving on from the analogy, there are a few more parameters to consider in automated execution than within the traditional process, such as the flexibility that exists on the exchange with regard to asking for better odds either to back or to lay and for bets to persist during in-play markets. Some elements persist with an analogy with the traditional process even so – the ability since the end of 2007 on Betfair to take a Betfair Starting Price, for example, as well as whatever price is available at the current market, or indeed an early price – just as we can with a traditional bookmaker.

Those observations aside, on the exchanges we soon enter a different world in terms of betting execution (as well as the obvious difference that we can also place lay bets). This is primarily in the possibility to place bets outside the market that are unmatched, and with certain conditions attached, and then to alter them after they are placed.

In that sense we can never be sure if we are "on" or not, unless we make a separate call to the market to find out, and act accordingly. Presumably, this would be the equivalent in a traditional betting process of a bettor constantly returning to the counter and asking the cashier if her bet had been matched or not, and therefore whether she could have her counterfoil back. "Sorry, come back later". Tracking unmatched bets manually on an exchange is no less tedious.

So, we will cover the use of automatic functions to enquire on both matched and unmatched bets placed as a result of our betting strategy. Once an enquiry has been executed, we have options for updating unmatched bets (i.e. asking for a new exchange

price, requesting bet persistence through in-play markets, or leaving to SP) and cancelling them, which we will also cover as part of an automated strategy.

As we can see in touching upon the issue of unmatched bets, the execution stage may be an iterative affair, as opposed to marking the end of our strategy. By definition, as discussed in the last section of Chapter 9, *Automating Trading Decisions*, this is always the case in trading, since more than one bet must be placed to qualify as a trade, and execution of any single bet will always be only one element of execution - unless it marks the closing of a position.

In any case, the mechanisms used for enquiring on matched and unmatched bets, updating and then cancelling bets are no different whether used within a trading context or, for example, in trying to obtain a qualifying price for a selection in a betting strategy which is consistent with the desired strike rate and strike price for that system.

However, we begin by concentrating on betting execution for our automated scenario, namely betting on outcomes.

In the simplest case, we want to get on at the current back price or lay price in the current market and stand the best chance of fulfilling the bet. In this case, we are continuing the example strategy we have been using throughout the framework, for betting on a oddsline. Thereafter, we will move on to the case of conditional bets and bets outside the current market (simply changing the input parameters supplied to the functions for placing bets). Finally, we consider the more complicated case of dealing with unmatched bets and closing out positions to take advantage of market fluctuations.

Betting Execution on Oddsline Example

Betting execution is the final piece of the jigsaw for the completion of our example oddsline strategy. However, most of the work has already been done for this particular strategy, since all the instructions needed to place each qualifying bet were printed out line by line to the file ***`selections`*** during the betting decision making process, as shown in *Example 9-1*.

All that remains to be done for the execution program is to pick up the file, read in all the variables from the ***`selections`*** file that were previously written by ***`bet_formation.pl`*** (i.e. the program name for *Example 9-1*) and pass these variables into the function for placing a bet.

Next we will cover the purpose of those variables, and the function used to process them and place our bet – essentially this is all the program in *Example 10-1* does, together with appending further information to a betting report.

The key call as far as the Betfair API is concerned is the aptly named **PlaceBets**.

The key parameters that we need to keep in mind for placing bets - and indeed all related functions such as **UpdateBets** that we cover later – are, in no particular order:

- Selection Id
- Market Id
- BetType
- Stake
- Price

The unique identifiers for **selectionId** and **marketId** will ensure the bet is placed on the correct selection running in the correct market. The other parameters are determined by our strategy and staking plan – being

- BetType, to determine whether we are taking on a 'L' (for **lay**) or binary 'B' (for **back**) liability,
- Stake, to two decimal places
- The Betfair price, using the exact format for one of the Betfair odds increments discussed in the Chapter 9 in the section *Changing The Price Dynamically*.

Additionally, for specifying whether the bet is for the exchange market, to persist during the in play market (bet persistence type), or the effective stake in the event of an SP bet (the same as the back stake if a back bet, or the total liability if a lay bet), we have, respectively:

- BetCategory
- Bet Persistence Type
- BSP Liability

Lastly, the **AsianLineId** is a mandatory argument for this function, but does not concern us for horseracing markets (so we simply specify a value of zero when making the call).

As with all the other stages in the framework, this part of the process can be tested and used independently provided there is a valid input file available. Indeed, no matter how we get to the point where we are placing bets - whether that be through years of research and creating an oddsline method from the ground up; data scraping from a tipster's webpage, or selecting all horses whose name begins with "Z" from the daily form dataset – as long as the listed parameters are simply presented for each selection in the right format, our betting execution program will not bother with their provenance, but simply execute such bets on the exchange.

Example 10-1: Executing Bets from a list of selections

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this example is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#

#       prerequisite modules to run this script
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
```

```
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";                   #productId is for Betfair's Free Access API

#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }

#    open files for reading in selections, writing out a report and record of bets placed
open(SELECTIONS, '/home/aeb/selections');
open(REPORT, '>>/home/aeb/betting_report');
open(STRATEGY_RECORD, '>>/home/aeb/daily_bets');

my $date = `date`;     #call to system time at start of program, alt = use localtime
$date =~ s/^\s+|\s+$//g;

print REPORT "
------------------------------  Bet horses
$date\n";

#Go through selections file and parse bet parameters on each line
my ($time, $course, $horse, $marketId, $selectionId, $price_asked, $betType, $stake,
$betCategoryType, $betPersistenceType, $bspLiability);


while(<SELECTIONS>) {

($time, $course, $horse, $marketId, $selectionId, $price_asked, $betType, $stake,
$betCategoryType, $betPersistenceType, $bspLiability) = split (/,/,);

#    clean all parameters ready for betting

$selectionId =~ s/^\s+|\s+$//g;
$marketId =~ s/^\s+|\s+$//g;
$price_asked =~ s/^\s+|\s+$//g;
$stake =~ s/^\s+|\s+$//g;
$betType =~ s/^\s+|\s+$//g;
$time =~ s/^\s+|\s+$//g;
$course =~ s/^\s+|\s+$//g;
$betType =~ s/^\s+|\s+$//g;
$betCategoryType =~ s/^\s+|\s+$//g;
$betPersistenceType =~ s/^\s+|\s+$//g;
$bspLiability =~ s/^\s+|\s+$//g;

#print "$selectionId, $marketId, $stake, $price_asked, $betType\n";

#Setup missing parameters for the PlaceBet call
my $asianLineId = 0;

#Set up variables for the return values from the call
my $bet_error = 'errorCode';
my $bet_ref = 'bets';
```

```
my $betId = 'betId';
my $bet_placed;

my @params= ($token,$asianLineId,$betType,$marketId,$price_asked,$selectionId,$stake,
$betCategoryType,$betPersistenceType,$bspLiability);

#make the call and return values to a hash
my %bets = place_bet(@params);

#print Dumper (%bets);          #to inspect returned data structure

#Check for errors and write reports

if ( !($bets{$bet_error} =~ /OK/) )
 {
   print REPORT "Failed bet placement:\n";
   print REPORT " $bets{$bet_error}\n"

          }
 else
 {
my $bet_placed = $bets{betId};

                      print REPORT "Following bets placed:\n";
                       print REPORT "$course, $time, $horse, $betType, $price_asked,
                       $stake, $bet_placed\n";
                      print STRATEGY_RECORD "$bet_placed\n";

}       #end else for printing out successful bets


        }       #end while reading in selections



#quick and dirty test for determining selections input
if ($time =~ /[0-9]/) {
print REPORT "Bet horses finished for $time\n\n"}
else {print REPORT "\"Selections\" file is not written or blank\n\n";
}
```

The first part of the code in *Example 10-1* simply logs in ready to bet and then reads in and cleans the variables from the *`selections`* file that were already written out by *Example 9-1*.

We are agnostic to the input, which could be from any source, as long as it is provided in the correct form – meaning this example can more easily be adapted for different or non-oddsline based betting strategies. The program simply requires the inputs, hence why this stage is a question of execution only and can be joined to any method. All the parts of the program relating to reporting `REPORT` are specific to our example strategy, and can safely be ignored or adapted otherwise.

In Part 3, we combine the decision making and betting execution programs for this particular example strategy, thereby making some code – which is effectively repeated in both examples - redundant. However, maintaining discrete programs when putting strategies together means that each stage in the framework can be tested individually.

By scheduling the programs in the framework, including this one, to run sequentially, the example program will place any bets received like a relay runner, accepting the baton from the program that preceded it, provided that the bets received are in a certain format, just as the baton must be passed in a certain manner in a relay race – although

perhaps that's enough about batons for now. An example of the output from running all the framework programs, including this one, is demonstrated in the first Chapter of Part 3, *Automated Strategies in Practice*.

The betting function in *Example 10-1*, takes parameters for one bet at a time, enabling us to open up a loop for reading in selections one at a time, and betting on them sequentially. However, it is worth noting that the **PlaceBets** API call itself is designed to read an array of up to 60 multiple bets at one time.

The call can easily be altered to accommodate reading an array of bets (as opposed to one) for additional speed, which would be of particular importance in a trading strategy. It's worth noting that we could alter our program to accommodate that method with a little extra work to save the bets we want as an array or arrays, then looping through each bet element, extracting the details and passing that into a revised function which expects an array of arrays as one of its arguments.

For convenience however we will stick with the single version for now. Not only is it simpler from a programming point of view, it is also relatively efficient when making only one to a few bets, as per our examples. How this function is executed in practice all depends on the user's strategy.

The rest of the code leads up to using the Perl **`place_bet`** function which implements a single call to **PlaceBets** for one set of bet details, first preparing all the arguments for the function discussed at the start of the section, thus:

```
@params=($token,$asianLineId,$betType,$marketId,$price_asked,$selectionId,$stake,$betCate
goryType,$betPersistenceType,$bspLiability);
```

We can iterate over this function within the context of a loop to place all required bets, or modify the function to accept an array of bets from the program, although the code is simpler in the single, looping case.

Then, for clarity, passing the arguments to the function and returning the return values to the hash `%bets`.

```
my %bets = place_bet(@params);
```

The next section deals with the return values within the context of the program.

Collecting the Bet Identification

We split the rest of the code walkthrough in *Example 10-1* into this section to focus on a critical return value from placing a bet, namely the bet identification number or **betId** in terms of the Betfair XML element returned.

The bet identification number is the unique reference for the bet that has made, whether that bet is matched or unmatched, or placed at SP. Any subsequent action on the bet

from an automated point of view will require the bet identification number – whether to amend it (via update bets), cancel it, or to enquire upon its result.

The **betId** is collected within the hash that is returned by calling our **PlaceBet** function – we just have to extract as below:

```
my $bet_placed = $bets{betId};
```

At this point the **betId** is available for reuse within the program (e.g. checking on unmatched bets and updating the bet with new parameters) and/or for writing out to a file or database.

Example 10-1 uses the **betId** to reconcile profit and loss and for record keeping purposes, thus all bet IDs are concatenated to a file (as with the double arrow writing to the filehandle `STRATEGY_RECORD ">>`), so that each bet (or iteration of the betting program) for a given strategy adds its details to the file. Once written to a file, the **betId** is the critical variable to use in order to keep track of the success or otherwise of a particular strategy after a market has been settled, and one which we use for the purpose of record keeping in the next chapter.

Note that *Example 10-1* does not check for unmatched bets. This keeps the program simpler, but also for this strategy (which plays in UK horseracing win markets to £5 at the current back price in 5 minutes before the offtime) there is little risk of being unmatched as we show in testing during Part 3.

However, best practice for building any automated strategy means such tools for ensuring the bet is matched should be at our disposal. In particular, if using a strategy which places bets outside the current market (either on criteria of price or current volume), or for a trading strategy (where for example it is essential to know if the first leg of the trade has been matched before executing on the second leg), checking for unmatched bets is a prerequisite to the strategy. We therefore cover this in the next section.

Placing unmatched bets outside current market prices or volumes

In testing strategies to live stakes during Part 3, we can see that following a policy of taking the available back price to smallish stakes (e.g. anywhere up to £10 per bet) in a tight market (i.e. with a good overround and high liquidity, such as during the last 10 minutes prior to the off), ensures a very high chance we will be matched at the back price - if the back price is the price we want.

However, outside these tight constraints, the likelihood of being matched outside the current market is dependent on the strategy's ambition for one or both of either:

- **Higher Stakes** than the current market offers
- **Better Prices** than the current market offers

Our ability to get matched will be inversely related to that ambition.

In both cases, the longer the time to the market offtime, the more chance there is of profiting from market volatility – since closer in time, the next value is statistically likely to be nearest the previous value. Time helps move values far away from their current points, but asking for both higher price and volume which is outside the current market range, it will become increasingly difficult to get matched the further we travel away from current market liquidity.

Where the betting strategy requires a marginally better price or volume at the front of the market, or is playing illiquid prices to small, speculative amounts, betting execution will not only involve placing bets, but checking on outstanding bets (i.e. that have not been matched), and amending (i.e. updating) or cancelling bets. The exception to this rule is the Betfair SP market; where bets cannot be cancelled or amended once placed.

In all cases, we have additional options to switch bets when updating them to either the SP market or to add an additional market to the bet's life, i.e. to let it persist during the in-play or in-running markets.

Firstly, however, programs need to be able to check the status of bets and act upon that status, for any strategy which:

- Is likely to generate unmatched bets (a potential risk with any strategies, but an obvious risk where speculative bets are placed)
- Explicitly needs to give the option to revise bets downward or upward from the asking price at a subsequent stage
- Includes trading elements, such as a second bet dependent on the first bet

Checking on outstanding (unmatched) bets and positions

A number of calls exist in the Betfair API which enable the bettor to check on various aspects of outstanding bets and positions. There are also numerous Betfair API "Lite" services outside the context of the Free Access API subscription level.

However, within the context of both the Free Access API and subscription API, **Get Matched and Unmatched Bets** is a useful ally, and it is this service and corresponding function that we will focus on in this section. (N.B.: The full list of API services is detailed at the start of Chapter 4 and a comprehensive reference to the inputs and outputs for each can be found in the *Betfair Sports API Reference Guide*, as discussed in Chapter 4.)

The Perl function retrieving details on Matched and Unmatched Bets used in the example library - **`get_mubets`** - provides two ways of capturing matched and unmatched bets – by **marketId**, or by **betId**. (It should be noted that data request charges are higher in the Free Access API for multiple calls made for bet IDs which may travel across many markets, than for market IDs, for single markets). There are various other parameters available for accessing matched and unmatched bets, including parameters to access bets according to whether they are matched, unmatched or both.

We hard code the options in the example function so that both matched and unmatched bets will be returned according to the arguments (i.e. either **betId** or **marketId**) supplied to the function.

If only the **betId** is supplied, then only the matched and unmatched information relating to the bet we specify will be returned. In this case, the **marketId** should be set to a value of zero – as shown in *Example 10-2*.

Specifying the **marketId** (in which case, the **betId** should be set to zero) will return all bets from the market, matched and unmatched, including any bets which have been updated, for the market in question. What we do with the returned bet and status information will depend upon the strategy that created all the bets there in the first place – we cover some possibilities after the example code for retrieving them.

The data structure itself that is returned by **`get_mubets`** is a hash. The hash keys refer to the variables in the header response (common to all functions), but the elements we are particularly interested in are referenced by the key **`results`**, as below:

```
$mu_bets{results};
```

This represents a further data structure holding any matched and unmatched bet details (by market), or the particulars of one bet (by **betId**). The scalar variable we use to save this key is an anonymous array of hashes, with each hash being a set of different bet details relating to one **betId**. In the case of one bet only being returned, our subroutine (listed under **`sub get_mubets`** in the example library) forces any single bet to be an array also, so that all variables pertaining to the bet can be accessed consistently – that is, by dereferencing the array and then referring to the hash keys for each value, as in *Example 10-2*.

Example 10-2: Retrieving Bet Status by BetId or MarketId

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.                  #
# The code from this example is provided under the terms of the Artistic License 2.0        #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code           #
#-------------------------------------------------------------------------------------------#

#       prerequisite modules to run this script
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";                   #productId is for Betfair's Free Access API

#       login to the Betfair API
my %login = login($username, $password, $productId);
```

```
my $token = $login{sessionToken};
my $login_error = $login{errorCode};

if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }

#  A betId or marketId is normally supplied to get_mubets within the context of a program
#  Here we test the function manually with any given betId or marketId
#  If only marketId is supplied, betId must be set to a value of zero - and vice versa
#  See also Examples 10-3 and 10-4, which are continuations of this code
#  For updating and cancelling bets

#my $betId = "0";
my $marketId = "20852151";     #test by hard coding a marketId where a bet exists

my %mu_bets = get_mubets($token, $marketId, $betId);
#print Dumper(%mu_bets);

my $results = $mu_bets{results};

#print Dumper($results);
my ($mu_status, $mu_size, $mu_price, $mu_betType, $mu_persistence, $mu_betCat, $mu_bsp,
$mu_selectionId);

foreach my $ref ( @ { $results } ) {

$betId = $ref->{betId}{content};
$mu_status = $ref->{betStatus}{content};
$mu_size = $ref->{size}{content};
$mu_price = $ref->{price}{content};
$mu_betType = $ref->{betType}{content};
$mu_persistence = $ref->{betPersistenceType}{content};
$mu_betCat = $ref->{betCategoryType}{content};
$mu_bsp = $ref->{bspLiability}{content};
$mu_selectionId = $ref->{selectionId}{content};


print "$mu_status, $mu_size, $mu_price, $mu_betType, $mu_persistence, $mu_betCat,
$mu_bsp, $mu_selectionId, $betId\n";

if ($mu_status eq "U") {

#do something with each unmatched bet .

#e.g. enquire on the price, update it (covered in next section), etc.

}; #end if matched status is unmatched

}       #end foreach reference to a betId or betId within market
```

The code we are concerned most with above comes after making the function call and returning the anonymous array of hashes containing all the bet information, which is stored to the variable **`$results`**.

The body of the example simply retrieves and prints out the values of variables that are needed when updating and cancelling bets.

```
$betId = $ref->{betId}{content};
$mu_status = $ref->{betStatus}{content};
$mu_size = $ref->{size}{content};
$mu_price = $ref->{price}{content};
```

```
$mu_betType = $ref->{betType}{content};
$mu_persistence = $ref->{betPersistenceType}{content};
$mu_betCat = $ref->{betCategoryType}{content};
$mu_bsp = $ref->{bspLiability}{content};
$mu_selectionId = $ref->{selectionId}{content};


print "$mu_status, $mu_size, $mu_price, $mu_betType, $mu_persistence, $mu_betCat,
$mu_bsp, $mu_selectionId, $mu_betId\n";
```

In practice, the first parameter we are interested in is the status of the bet – returning **M** for matched or **U** for unmatched. If the primary concern is with unmatched bets, then anything with the indicator **M** is of no relevance. In *Example 10-2*, we take no further action on matched bets, apart from printing out summary details, although of course we could go further, such as calculating position and average price in the case of multiple bets on the same race or horse.

Unmatched bets will return a status **U** to the **`$mu_status`** variable. In the example shown, we split out any unmatched bets to make a decision – which could be to leave the unmatched bet alone until it is matched, or lapses; to explicitly cancel it, or to amend it in order for the bet to be accepted on different terms and conditions.

To do this programmatically, we may set up a loop to constantly enquiry on the status of the bet. Such a loop could depend for its execution conditions on time remaining to the event start, with a series of actions to sleep, enquire on the **betId** or market ID, then sleep, enquire on the **betId** or market and so on, up to a certain point (let's say one minute before the off), when a decision would be finally be made (in programming terms), as to how to proceed. An adaptation of the type of construct we outlined earlier in Chapter 7 for looping over market prices at regular intervals would suffice for this.

Such a loop, enquiring on the status of unmatched bets, can be combined with an enquiry on market prices or any other API call, and logic built into the program to evaluate those prices.

To find out if the next bet (or only bet, in the case of one returned value) is matched or unmatched, we simply test for the matched status

```
if ($matched_status eq "U") {

#do something with each unmatched bet -
#e.g. enquire on the price, update it (covered in next section), etc.

}; #end if matched status is unmatched
```

Sometimes a bet may be partially matched, such as when a price or amount being requested is close to the market, or in a timing sense, if the market value we asked for was in the process of disappearing just before our bet was put onto the exchange. In all cases, some amount may be matched at the requested price but not all that was asked for.

In the case of a partially matched bet, the solution to test for whether or not the bet is unmatched (and therefore whether any action should be taken) is no different to a completely unmatched bet. The array holding the hashes of bets was forced to deal with single or multiple values, so if there is more than one value (as in the case of a partially matched bet) the test will return the part of the bet that is unmatched. In this

case, the betId will be the same as our original bet, although referring to a partially unmatched portion of it. The **betId** will only change if the terms and conditions of the unmatched portion are changed by updating it.

In all cases of unmatched bets, we can take the returned parameters and use them to explicitly update bets in many varied ways, or to explicitly cancel them.

Updating and Cancelling Bets

Following on from the previous section, let's suppose we built tests into our automated strategy for whether bets are matched or unmatched (which includes partially matched bets), and that the following scenario now applies:

- With one minute before the start of the race, a **betId** returns unmatched status. We collect market information on the selection using the betId to retrieve its details. The market information shows that the current price is some ticks below the price that we asked for. Next we repeat the "betting decision making" criteria for this bet.

Now, according to the decision making criteria, we have the following possibilities:

- Value still exists in the selection, albeit at a lower price than we originally wanted. According to the strategy, the program should therefore update the bet, or the partially matched bet, in order to get on at the current market price before the original bet lapses at the offtime.

- As above, we test for market prices, and apply the decision making criteria, but the current price is below that required by the strategy We therefore have a choice as follows:

    - Update the bet so that it can persist at the current price during the in-play market, taking a chance that we will be matched, (bearing in mind that we could also update any other aspect of the bet, including stake and price asked).
    - Update the bet and switch to SP, guaranteeing a match but not the price (note that on Betfair we can't specify a minimum price for switching a bet to SP, only when placing an original bet at SP)
    - To let the bet lapse
    - To cancel the bet

At this point we may well question, given the uncertainty on any bet outside current market parameters being matched, coupled with all the programming effort involved, exactly what benefit is involved in trying to "get on" automatically outside current market prices?

In such cases, the strategy can explicitly cancel the bet. However, there are strategies where all nuances in the scenario are appropriate, especially with minimum value requirements close to the current market, or with a strategy that may rely on persistence during the in-play market.

In such cases, the strategy will need to update bet details on any bet that is found to be unmatched, in a further attempt to match it according to specific criteria. Let's look at an scenario to illustrate this.

A bet has been placed for £10 on *Faasel* in the Gold Cup Handicap Chase at Cheltenham on December 14th 2007, with a back price stipulated of 19.0.

By running the enquiry for unmatched bets on this market (showing the variables mentioned in the print statement in *Example 10-2*), we can return the following type of output to illustrate the arguments we are using:

```
U, 10.0, 19.0, B, NONE, E, 0.0, 416818, 4085811477
```

The bet is **U**nmatched, for a stake of **£10** at a price of **19.0**; it is a **B**ack bet, with no persistence conditions attached, placed on the **E**xchange as opposed to at SP, with no SP liability, **0.0** (effectively the stake or lay liability for a bet placed at BSP) with a **selectionId** of 416818 and a **betId** of 4085811477.

Following *Example 10-2* to the point where we execute a loop conditional to finding any unmatched bet, let us say for the sake of argument that within that loop we make a call to **GetMarketPricesCompressed** to retrieve current market values for *Faasel*. This shows us that *Faasel* is currently trading at 15.5 to back and 16.0 to lay, but there is approx 5 times the money to all 3 places shown on the back side than the lay side.

In other words, bettors on the exchange are 5 times more willing to lay *Faasel* at this price than they are to back him, in the currently available to bet market at least.

There is a fair time until the market starts, so, coupled with weight of money, there is an increased chance that a price of 19.0 could be matched. If it is not matched before the off, the race type is a competitive handicap chase (high value and large field) of almost 3 miles in which there should be a strong in-running market due to the market conditions (strength of market, interest in market and duration of race), so there is plenty of time for *Faasel* to be matched at a slightly higher price than SP, especially to small stakes. Our program will therefore make the call to leave the bet price as it stands, but to keep it when the market turns in play, increasing the chance of being matched. We thus elect to automatically change ("update") the persistence type on the existing bet.

Using the library function for updating bets, we simply pass in all the existing parameters for the unmatched bet, except those that we wish to change (as with the website interface, however, it is not possible to amend both size and price on an existing bet).

Below we can imagine that the code is a continuation of *Example 10-2*, so that we will use the current values returned by the unmatched bet, simply changing the parameter that needs updating - here it is the "In Play" parameter, although any other parameter could be changed in the same way.

Example 10-3: Updating Unmatched Bets

```
#!/usr/bin/perl -w

#---------------------As Example 10-2 up to and including the if statement below

if ($mu_status eq "U") {

#do something with each unmatched bet .

#here we change all unmatched bets to status "Keep" so that they will persist in play
#Not a good idea unless the strategy merits it…

my $newPrice = $mu_price;
my $newSize = $mu_size;
my $oldPrice = $mu_size;
my $oldSize = $mu_size;
my $persistType = "IP";
my $oldPersistType = $mu_persistence;

my %updated_hash = update_bet($token,$betId,$newPrice,$newSize,$oldPrice,$oldSize,
$persistType, $oldPersistType);

#print Dumper (%updated_hash);  #check data structure for all possible values

my $new_BetId = $updated_hash{newBetId};
print "$new_BetId\n";

}; #end if matched status is unmatched

}  #end foreach $ref (start of loop not shown above, but is required in Example 10-2 of
       # which this code is a continuation)
```

In the example for *Faasel*, only one parameter in the call has been changed (i.e. `$persistType` **NONE** has been changed to **IP**). Thus the bet on *Faasel* will now persist into the In Play market if it is not matched by the time that the race starts.

For the scenario where we want an unmatched bet to be matched with relative certainty before the in-play market, the same function call for updating bets applies. Simply changing the bet persistence type to **SP** in the case of SP markets, or leaving it at **NONE** and changing the price or amount only in an attempt to be matched at the current market.

In each case, we also save the variables from our original request for retrieving matched and unmatched bets, and pass these to the **`update_bet`** function, as already shown. We can only update size or price in one call, just as with the website interaction.

In the example shown, the size of the unmatched bet comes from our existing strategy, so there is only a need to change price, and the condition of the bet, if required, relating to bet persistence.

Thus we have 3 strategies for updating bets hinging around the use of the **betPersistenceType**

- **NONE**: No bet persistence type, bet can be updated to request a new price (e.g. the current market price to be sure of getting on at the market) and amounts
- **IP**: "In play" persistence; unmatched bets (or the unmatched portion of a bet) remain in the market when it switches to in-play. If no other parameters (price or

amount) are changed, the same betId will be returned, otherwise a new betId will be returned reflecting the updated bet.

- **SP**: Converted to market on close starting price

A critical point to bear in mind for any use of the `update_bet` function which changes price or size of the bet is that a new betId will be assigned to the "updated" part of the bet. If this bet is itself unmatched, and we wish to change any further aspect of it, we will have to maintain and pass in the new betId. If bets continued to be partially matched, one original bet could thus spawn many bet IDs through any strategy to update them.

Although this hypothetical case has been discussed in terms of an interactive human decision, such logic can easily be embodied within the context of an automated strategy. For example, we could stipulate that for certain types of races (such as long distance chases) a price is always requested at N% above the current market price if there is some time to the off. Subsequently, according to some appropriate conditions, any unmatched bet could be updated as above, to persist during the in-play market.

Let's say that the bettor does not agree with any of the above, and changing the bet to bet persistence does not make sense. At the same time, leaving a bet outstanding that is unlikely to be matched is affecting their overall Betfair exposure. They want to explicitly cancel the bet.

To explicitly cancel an outstanding unmatched bet, the call is similar to update bets, the API call is simply **CancelBet**. To illustrate using cancel_bet in context, we can take our example code up to the same point in *Example 10-2* (where we have automatically retrieved details of unmatched bets) to explicitly cancel unmatched bets instead of to update them.

Example 10-4: Cancelling Unmatched Bets

```
#!/usr/bin/perl -w

#---------------------As Example 10-2 up to and including the if statement below

if ($mu_status eq "U") {

#here we cancel each unmatched bet according to the betId

my %cancelled_hash = cancel_bet($token,$betId);

#print Dumper (%cancelled_hash);
my $size_cancelled = $cancelled_hash{sizeCancelled};

print "$size_cancelled\n";


}; #end if matched status is unmatched

}  #end foreach $ref (start of loop not shown above, but is required in Example 10-2 of
       # which this code is a continuation)
```

Hedging, and taking a partial profit on a position

"Greening up" or ending up with a "green book", as the term has been coined, is typically associated with profitable trading activities (laying low and backing high or vice versa) on either one or more runners in a race. This could result from trading an individual runner. It could also result from attempting to create a book by backing (dutching) or laying every runner in the field to a favourable overround.

As those who have used Betfair with any frequency will know, "greening" out or up is so called since the profit figure displayed beneath each runner on the website interface, showing profit or loss in the event of that contender winning, shows in green for all runners, as opposed to red for some and green for others. The bettor with a green book knows a profit will be achieved whatever the outcome of the event.

Theoretically, this is the ideal position for a trader to be in before any race, since he has made a profit before the race begins. By contrast, the logic in betting on outcomes is that we are prepared to wait for the outcome in the expectation that the prices available on our selections do not reflect their true chance.

The question is, at what point does a price change mean it makes sense to cash in on that bargain price if the price changes before the event (or even during the in-running markets)?

We could argue that there are certain instances (i.e. when prices significantly contract or lengthen before any race) when any bettor on outcomes should also contemplate thinking like a trader, and that this should therefore be accommodated within any betting process.

For example, let's say our oddsline has determined that a contender's "true price" or probability of winning is 8/1 (or 9.0 to exchange decimal odds). On visiting the market and executing the oddsline strategy, we find that the actual price on this contender is 10/1 or 11.0, a healthy 20% higher than our estimate. The selection therefore constitutes a bet, which we place some time before the off. However, at the off, the price has contracted to 6/1 or 7.0, now more than 20% below the price that we originally considered to be true value, and 40% less than the price we managed to obtain. For the trader, locking in a profit at this point is a "no-brainer". For the backer or layer who was intending to bet outcomes also, hedging the bet or taking some profit at this point can also be a logical option, entirely consistent with an oddsline strategy, which is after all about identifying value.

In terms of implementation within the automated framework, this means executing our first bet, and then saving the bet ID in some persistent format (a file, a database, or in-memory if the program is kept alive), so that we can act upon a later price for the runner and place a subsequent bet, either to back or lay, which locks in some profit.

The exact profit or hedge required will depend on the strategy.

- We may simply decide to let the bet ride until its outcome, and that the contraction in market price confirms the value obtained, therefore there will be no hedge.

  - So for the no hedge case, for a £10 stake at the back price of 11.0, winnings remain £100 and should the bet lose £10 is at risk.

- The objective could be to hedge if the price contracts but not to lock in a profit, rather to be left with a "bet to nothing", where the back stake is covered and a potential profit remains so that there is a no lose situation.

  - The bet to nothing will simply involve laying the selection for the same stake at the new price.
  - So for a £10 stake on the example shown, backing at 11.0 and laying at the later contracted price of 7.0, the profit would be £40 (without commission), with zero risk if the selection lost.

- The objective could be to lock in profits whenever price movements of this magnitude (e.g. 30% plus) occur in favour of the oddsline strategy. Locking in profits on the whole market is equivalent to the same method to be followed in a trading strategy where we want to "green up" the book after a price change in the trader's favour, whatever the size.
  - Thus, in terms of our example, we place the original back bet of £10 at 11.0, and then leave it standing until the strategy revisits the market to determine the later price.
  - The market price has contracted to 7.0, 40% less than the price we obtained on the selection.
  - The automated rules for "greening up" the book kick in, so we must place the appropriate lay bet in order to guarantee an equal profit on the market in the event of any of the contenders winning.
  - The calculation of the stake to place at the current price (or desired price, if setting a hedge outside the current market) is:
    - **Lay stake = (back price/ lay price) x original stake**.
    - = (11/7) * 10 = £15.71.
  - Working it through on a bet by bet basis:
    - Should the selection win, we will win £100 from the original bet, and lose £94.26 from the lay bet, leaving a profit or £5.74.
    - Or, should the selection lose, we lose £10 from the original bet, and win £15.71 from our successful lay bet, leaving a profit or £5.71 (the 3p difference being down to rounding). Thereby guaranteeing a profit of £5.70 (without commission) whatever the outcome.

In the opposite case, to equalise profits for a "lay to back" strategy where a price lengthens, the terms lay and back above can simply be reversed.

As we can see, trading or hedging strategies of this nature generally involve straightforward maths that can be easily coded for automation. We can program rules based around the extent of price contraction in order to determine when and when not to hedge. In such instances, a more difficult question for the bettor is less how to hedge, but at what price level it is more profitable to hedge our bets than leave to them to run.

Chapter 11: Automated Record Keeping

The final stage of the framework is not mission critical to the process of placing an individual bet to implement a betting strategy - but it is deemed best practice for any betting strategy, manual or automatic, for a number of reasons.

Only by keeping records can we hope to assess the success or failure of any strategy. In the Betting Execution process, we deliberately went out of our way to record the bet ID of all bets placed for this reason – to automate the record keeping process. We also captured errors and output from calls made at the time the bets were executed – here we were more concerned with mirroring the manual process of analysing bets and betting strategies.

However, in a manual betting process, record keeping is often seen as a medicine that should be taken but is rather avoided (especially in losing runs), since it involves the tedious, uncreative and time consuming task of typing or writing up records after betting activity, and a consequent diminishment of fun.

At a minimum, any bettor hoping to implement profitable strategies clearly needs to know her profit and loss on each bet, and cumulative profit and loss. The good news is that, at least for individual accounts, electronic bet placement has greatly simplified the task of record keeping, be it for manual or automated bets, with each bet being recorded, as well as total deposits and withdrawals to exchange accounts. The Betting History, Betting P&L and Account Statement facilities on the Betfair website interface also enable easy downloading of betting histories to spreadsheets for further analysis on a manual basis.

However, knowing what the profit and loss figure is on its own is only a small piece of the story in using betting history to the bettor's future advantage. Unless it is clear why each bet has been struck, there can be no improvement based upon betting history.

On a more mundane level, if we do not automatically record and store bets according to which strategy they relate to, then it is impossible to assess success or otherwise on a strategy by strategy basis, and no meaningful information can be derived. So mixing up profit and loss on all bets together for radically different betting strategies may tell us our overall Account profit and loss, but nothing that we can use to help us analyse and improve.

We can therefore start to look at the value added (and more interesting) part of the process of record keeping (i.e. analysis) by automating the capture of bet IDs relating to individual strategies *when the bets are made*. Subsequently, further details of the bets can be retrieved using various API functions, and full information can be recorded electronically for future access and analysis, for example in a database.

This data will be a combination of bet details (horse, race, price and so on), why the bet has been placed and under what conditions (in other words, details of the strategy to which the bet relates and the decision criteria), and the result of the bet (in terms of profit and loss).

What we can do with such data, once we have it, is itself the basis of developing many aspects of a betting strategy further – from staking, decision making criteria, and the basis of the system itself – for example, looking at expected versus actual returns.

Capturing settled bet details

The ability to analyse betting strategies relies on capturing details of each bet executed within the strategy. The process starts with recording details of bets that have been made as part of a strategy, during the betting execution phase, and ends with capturing and recording details of each settled bet when the event has finished.

We first need to record the bets placed by any strategy at the point of execution, writing a record of these to a local file or to a database, along with any other information from the strategy – such as the oddsline tissue, and other criteria used to make a betting decision - which we might wish to keep for subsequent analysis of a betting strategy.

Secondly, we need to retrieve details relating to settled bets, fundamentally the profit and loss, together with any other information available from Betfair – such as the requested size, the matched size, the time the bet was placed, and so on - which is required for subsequent analysis.

Thirdly, we write those records to a database, so that we can track ongoing profit and loss per betting strategy and later use the database to analyse the results of any betting strategy at will.

Probably the most convenient Betfair API service for retrieving profit and loss with other summary details (supplying our saved **betId** as an argument) is the **GetBet** service. We use this function for convenience when storing strategy details in Part 3, *Automated Betting Strategies in Practice*, since bets can be retrieved and stored rapidly without time delay. However, this is not be used as the principal example, since this particular function is only available through the subscription API, or may be available within the context of a software vendor's supported application.

Although readers may eventually wish to go down the route of using subscription services, we will concentrate in this section on another way of skinning the cat by using the Free Access API service, **GetAccountStatement**, in order to retrieve the same details. Retrieving details for an existing set of bet IDs is slightly more painful by going down this route than by using **GetBet**, although we end up with the same results. In any case, the function itself is worth using from a general point of view for record keeping purposes, since there are other useful parameters available that are not within the **GetBet** service, such as the individual account holder's commission rate and commission deductions per bet. **GetAccountStatement** is in some ways the superset service to use for record keeping since all details relating to bets placed within an account are available, mirroring all the details usually available by looking up "My Account" from the Betfair website interface.

Retrieving Betting Records by Betting Strategy

Here we are primarily concerned with sifting through all the records within a Betfair account in order to find those bets automatically placed by a certain betting strategy, extracting them, then writing them to a database, so that we can assess the performance of any given strategy.

For this example, let's therefore assume that we have a file to which a list of betIds has been written, one **betId** per line each relating to an individual strategy. In fact, such a file was already generated within the context of the framework in *Example 10-1* during the Betting Execution process, (saved as a file called `daily_bets`, with the file handle, `STRATEGY RECORD`).

Further, we will assume that the purpose of this file is to supply **betIds** for our betting record program at the end of every racing day, including P&L for a strategy. This type of program would therefore be run as a scheduled task (e.g. a **`cron`** job in Linux) as per the discussion in Chapter 7.

The arguments for our Perl implementation (**`get_account_statement`**) of the **GetAccountStatement** API service are the obligatory **session token**, the **start date** and the **end date** for all the bets to be retrieved.

For the example, we make some general assumptions that such a program is set to run at the end of the day, and to retrieve all the current day's bets (no longer timeframe). To do this, we therefore specify a start date of the current date, with a time start of 00:00 hours, and an end time of 23:59 hours for the current day. This will pick up all daily bets placed in the account specified.

The bettor will have their own requirements for what data is recorded and in what format - for the example we use a simple database, the table for which should be created first, as shown in the following example:

Example 11-1: Generating a MySQL table to record betting strategy performance

```
#CREATE results table of bets to record the strategy PD_oddsline_bets in book, can be
#generic for any strategy, simply need to insert the relevant strategy name

DROP TABLE IF EXISTS bets;

#@ _CREATE_TABLE_
CREATE TABLE bets
(
        date                    date NOT NULL DEFAULT '0000-00-00',
        marketId                INT(11) NOT NULL DEFAULT '0',
        betId                   BIGINT(12) NOT NULL DEFAULT '0',
         strategy_name           VARCHAR(25),
         selection               VARCHAR(25),
        avprice                 DECIMAL (6,2),
        betsize                 DECIMAL (7,2),
        PandL                   DECIMAL (7,2),
        PRIMARY KEY (marketId, betId)
)
#@ _CREATE_TABLE_

#The script to create the table can be executed at a shell prompt as below:
#  mysql < create_bets_table.sql
```

The table can be used generically (i.e. to record the performance of any betting strategy), simply by altering the strategy name. In the context of this example, we will use the strategy name **`PD_oddsline_bets`** to refer to the example strategy started in Chapter 6, which has been used to illustrate the framework.

The requested bet size is a value which we would normally want to include in the database table since a simple comparison of the requested bet size and matched size for each bet can identify missed bets and demonstrate how easy it is for our betting strategy to "get on". Using **GetBets** we can retrieve such values at this point in the framework.

However, once again proving that there are usually many ways to achieve different objectives for automation, this value can also be captured and recorded without using **GetBets.** Simply, it can be written to a file or database at the time when the original **betId** is recorded (although it creates more dependencies in earlier programs), since the requested bet size is of course the same as the stake stipulated in the betting execution program.

Once the database table is created, we are ready to schedule a script such as the one below to pick up bet details every night after racing and record them for a given strategy, as discussed earlier:

Example 11-2: Retrieving betting records for an individual betting strategy

```
#!/usr/bin/perl -w

#-------------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.                  #
# The code from this example is provided under the terms of the Artistic License 2.0       #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code           #
#-------------------------------------------------------------------------------------------#


#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;

my $year = ((localtime)[5] + 1900);
my $month = ((localtime)[4] + 1);
my $day = (localtime)[3];
my $date_today = sprintf ("$year-%02d-%02d", $month, $day);

open (BETS, '/home/aeb/daily_bets');

my @strategy_bets = (<BETS>);


#       login variables
my $username = "username";
```

```
my $password = "password";
my $productId = "82";           #Free Access API access code

#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};


if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login successful";
 }

my $dbh = DBI->connect("DBI:mysql:autodb", "username") or die ("Error:  $DBI::errstr");
#substitute your database and user credentials

#substitute your betting strategy name
my $strategy_name = "PD_oddsline_bets";

my $start = $date_today."T00:00:00";
my $end = $date_today."T23:59:00";

my %statement_hash = get_account_statement($token,$start,$end);
my $items = $statement_hash{items};
my $statement_betId;
my $betSize;
my $running_pandl;
my $pandl;
my $market_description;
my $winLose;
my $avprice;
my $selection


#print Dumper(%statement_hash);  #to list all variables it's possible to extract

foreach my $item ( @{ $items} ) {

$statement_betId = $item->{betId}{content};
$betSize = $item->{betSize}{content};
$pandl = $item->{amount}{content};
$market_description = $item->{fullMarketName}{content};
$winLose = $item->{winLose}{content};
$avprice = $item->{avgPrice}{content};
$selection = $item->{selectionName}{content};
$marketId = $item->{eventId}{content};

#if retrieving all daily records without specifying specific bets, uncomment
#unless ($market_description =~ /Transfer/) {$running_pandl += $pandl};

foreach my $strategy_betId (@strategy_bets) {

if ($statement_betId == $strategy_betId) {$running_pandl += $pandl};
if ($statement_betId == $strategy_betId && $betSize>0) {

#write bet details to database

my $sql = qq(INSERT INTO bets VALUES
('$date_today', '$marketId', '$strategy_betId', '$strategy_name',
'$selection', '$avprice', '$betSize', '$pandl') );
my $query = $dbh->prepare($sql);
$query->execute;

} #end if statement Id is the same as strategy Id
```

```
} #end foreach strategy Id

} #end foreach statement Id

#capture and store the net PandL for the strategy if required, in a separate table if
required
print "RUNNING_P&L = $running_pandl\n";
```

Assuming the account in question is used for more than one betting strategy, our task is to match all betIds stored in the file relating to our strategy and compare them against all the details retrieved from the **GetAccountStatement** service, processing and storing such details whenever there is a match.

First we save the betIds gathered from the strategy during the day, as follows:

```
open (BETS, '/home/aeb/daily_bets');
my @strategy_bets = (<BETS>);
```

Then login and make the call to **GetAccountStatement** as follows

```
my %statement_hash = GetAccountStatement($token,$start,$end);
my $items = $statement_hash{items};
```

The bet details are returned as an anonymous array of hashes, with each hash being an "item" representing a **betId** and corresponding bet details.

We then save the variables shown in the script, and loop through each item returned, gathering bet details. One "gotcha" to watch out for is any items which contain duplicate betIds in the array – this occurs for any betIds which have returned a winner.

In such cases, we have one betId which returns the gross PandL for the bet, and another which returns the commission deducted from the bet. In order to monitor the success or otherwise of the strategy, the example returns gross PandL, since commission rates can vary, but we can consistently compare gross sums and then captures separately the actual PandL (i.e. the gross PandL minus the commission) for the strategy on the day, so we also have a view of actual PandL.

Another potential "gotcha" to watch out for are other items which do not relate to bets, such as transfers – we don't want to count money taken from the bank account as winnings!

```
unless ($market_description =~ /Transfer/) {$running_pandl += $pandl};
```

Our program finishes the loop, extracting all details for each item, then writes out the final values for each to the database table shown earlier. When the whole loop is done, we are also able to calculate profit and loss on the day for the entire strategy, which we print out at the end of the program.

```
print "RUNNING_P&L = $running_pandl\n";
```

The running P & L calculations can also be omitted from the program, and done at the database level, both for gross profit and profit after commissions, provided of course that these values are captured and stored.

Using the **GetBets** call, the program to retrieve bets is reduced in size and complexity. Assuming we get to a point in the program where we have a file containing all bet IDs, such as *daily_bets* and read that in as per *Example 11-2*, up to the point where the `@strategy_bets` array exists, retrieving bet details using **GetBets** can be done simply by looping through the array and applying the function to every bet in the array, as shown in *Example 11-3*:

Example 11-3: Using GetBets to retrieve settled bet details

```
#!/usr/bin/perl -w

#-----------------Copyright Colin Magee 2008---------------------
#----------One example using a subscription service as an alternative to Example 11-2
#----------….But ending up with the same results…

#----------Code omitted for:
#----------Load modules,
#----------Login to API to retrieve session token
#----------Open file for betIds as per Example 11-2
#----------save the file as an array, in @strategy_bets

foreach my $bet (@strategy_bets) {

my %bet = get_bet($token, $betId);

my $getbet_error = $bet{errorCode};
$token = $bet{sessionToken};

if ( !($getbet_error =~ /OK/) )
 {
   print "Failed GetBet:\n";
   print "$getbet_error";
 }
 else
 {
        print "GetBet call successful\n";
 }

my $marketId = $bet{marketId};
my $selection = $bet{selectionName};
my $avprice = $bet{avgPrice};
my $reqsize = $bet{requestedSize};
my $matchsize = $bet{matchedSize};
my $PandL = $bet{profitAndLoss};

print "$date, $marketId, $betId, $selection, $avprice, $reqsize, $matchsize, $PandL\n";

#----------write bets to database as before and finish
```

Extending the examples for bet analysis

Generally, the basic information for recording profit and loss, as well as details relating to the bet placement strategy, will be constant in any capture of settled bets. The exact implementation, depending on the programs and reports being used by the bettor, as well as the level of detail ultimately required for betting strategy analysis, will of course vary.

There are many ways we can retrieve bet placement details, to include the **betId**, at the time of execution. The program can write out captured bet IDs to a database

immediately, for example. However, we still need to capture settled bets and write to the database again on a subsequent visit. In *Example 11-2*, we worked on the basis that a bet ID had been explicitly saved to a separate file. However, we can cut out the file of bet IDs and reuse the program or betting report generated for each strategy – provided that the bet ID is always included within the context of the report. This is the case with the example strategy, in that the **betId** is also recorded within the file `betting_report` in *Example 10-1*, so that file alone can be used for retrieving bet details, avoiding the need for duplication and providing the possibility of capturing richer information on the betting strategy at the same time (i.e. by storing other details contained within the file `betting_report` to a database record).

One way of doing this after racing has finished is by parsing the text in betting reports so that our end of day program captures all lines which include a searchable term such as "BetId" or "Bet placed:", alongside any other information we know we will want to write to the database at the time of bet retrieval. Once the bet IDs are captured, we can proceed as before (i.e. writing out the bet details to a database), but with the addition of any other summary details, captured at the same time, from the betting report.

For example, in an overlay betting strategy, as well as capturing the bet IDs, we could capture the percentage size of the overlay that was taken at the betting decision making stage. To derive this, we would calculate percentage overlay – which should ideally relate to the size of the bettor's advantage – by using the tissue price and the actual price and write it out as part of our betting report so that this data is included in betting records and reports (as per *Example 9-1,* for the file `betting_report`). Subsequently this data can be written to a database so that profitability can be analysed with respect to the size of the overlay achieved.

Combining betting records with other captured data in a database, we can also start to build more complex queries in order to analyse our data.

For example, since we capture the **marketId** when retrieving settled bets, we can join queries from the table **`bets`** on the **marketId** field, together with the **`races`** database table (created in Chapter 5, *Betting Process 101: Daily Event Data*) to return other variables, such as the race description, or race type, in order to analyse the betting strategy performance by race type. E.g. Is the strategy significantly more profitable over flat than jumps? Over hurdles rather than chases? In handicaps or listed races, and so on.

The above simply considers the case of using the information we have already captured from the exchange, as it applies to bets placed alone. However, if we extend the net further, and combine betting strategy information with parameters from a racing database, we have the potential to go beyond intelligence on the current strategy and test alternative strategies, seeing "what would have happened if" the criteria were different.

For example, our Postdata oddsline strategy bet all selections which were overlays in the top third of the field, that is to say the third of the field ranked from the greatest to the least winning chance. However, this choice was not based on analysis of previous results. If we bet the top third of the field, for overlays only, we could potentially have a

day's betting without a winner. How do we refine the strategy with no information about what does actually work, or where the strategy is failing?

As we will see in Part 3, *Automated Strategies in Practice*, perhaps an oddsline has some predictive power but prices inaccurately. Perhaps pricing is more or less accurate for the top ranked selection but more inaccurate for subsequent rankings. The only way to assess and correct pricing is to therefore assess the prices (and thus probabilities) given by the oddsline against the actual performance.

Likewise, we can assess the accuracy of the overlays against market prices, and pinpoint where the greatest returns are to be found throughout the oddsline, rather than with the particular betting strategy currently in use. Querying results from a racing database such as *Smartform* can be done automatically, so that programs can be written to run on an automated basis comparing results to the oddsline. This is the start of that refinement process and improving the oddsline on a continual basis.

Above all, it is imperative to scrutinise the database of betting records once we have it, adding in more meaning by using other data elements where it makes sense. We will scratch the tip of the iceberg in Part 3, but enough to show some of the possibilities into gaining insight on where betting strategies are working and where they are not – meaning the automatic bettor can alter stakes, modify strategies, pull the plug on a strategy or introduce new strategies with some confidence, based on quantitative analysis of the data at hand.

Whilst the automatic bettor can save time on bet selection and placement through the day, and even in generating standard reports on settled bets, there is good argument to suggest that time spent studying such data, or on producing further analysis once bets are settled –whether that be a daily, weekly or monthly activity, depending on the strategy - is an essential part of the automated process.

This final facet of the betting framework therefore leads us nicely into Part 3, where we review strategies implemented on a "live" basis, and subsequently use the automated record keeping techniques described in this Chapter to assess and improve them.

PART 3: AUTOMATED STRATEGIES IN PRACTICE

Chapter 12: Bringing the automated framework together

In the next two chapters, we combine all the stages of the automation framework detailed in Part 2. In this chapter, we demonstrate and test automation of an example betting strategy and, in Chapter 13, assess and improve upon the implementation. The betting strategy is based upon the oddsline originally put together in Chapter 6.

The primary objective of the test is to assess how well the mechanism for generating oddslines and betting them automatically works in a live environment. Based upon that judgement, we review and refine the mechanism. However, at this stage we are not concerned with testing the merits of any particular oddsline itself, including the one used for the example, rather that everything works as we intended it to for betting the oddsline automatically. As such, this mechanism can be adapted to work for any oddsline strategy. In fact, in Chapter 14, we will take a different oddsline and set it to work using the same mechanism, adjusted for improvements.

In this chapter therefore, we are running a "live test" on the framework, in the sense that a real account and real funds will be used – resulting in real profits or real losses, albeit to small stakes, during the test.

Prior to facing up to the real world of risk that any betting entails, however it is portrayed – automatic or otherwise, be it on horses, sports, or financial instruments - we will say a few words in the next section about testing on the framework in general.

Testing The Framework

Testing is a theme which will recur over the next few chapters, as we look at improving the implementation and improving strategies whilst limiting risk. In this particular section, we are concerned with emphasising the fact that all the elements within the framework can be tested individually prior to running the whole strategy live, or "in production".

The process we have followed in putting together the framework is to trace a manual process, define the stages within it, then work on automating each stage, with a view to quantifying the inputs required and the outputs that will be needed for subsequent stages.

Each program illustrating stages in the framework has been written so that it can be tested interactively, from the command line, for one event. This means we can execute each program, review the output, and reuse it for the next stage in the framework, testing subsequent programs manually if need be.

The stages for each automated betting strategy may vary, but by going through a process of defining each strategy, splitting the tasks up into their smaller, atomic parts, and writing code (and testing) for each stage, the room for error is reduced when it comes to putting all the pieces together to work as a whole.

Before running a live test, we can therefore go back to the different chapters of Part 2 and individually test:

- Producing a daily events schedule
- Generation and formatting of the oddsline itself
- Automating scheduled tasks
- Retrieving market prices for a particular event
- Automating betting decisions by using prices and the oddsline
- Executing bets

If any part is not working, the cause can be diagnosed at an atomic level until the bettor is satisfied with it. At no point in the process is there any danger of risking capital until testing bet execution. In fact, there is no need to risk cash even at that point in order to test that the implementation works correctly. For example, the betting execution program in *Example 10-1* takes for its input a file produced by the betting decision making program shown in *Example 9-1.* Since we split the stages up, we can run the program on its own and test that the ***selections*** file is generated as expected, as in the example output below:

```
2.15, Fontwell, Mister Pink, 20849970, 1430682, 14.5, B, 2.00, E, NONE, 0.0
2.15, Fontwell, Iron Maid, 20849970, 1409367, 9.6, B, 2.00, E, NONE, 0.0
2.15, Fontwell, Dragon Eye, 20849970, 2324934, 12.0, B, 2.00, E, NONE, 0.0
```

This gives all the parameters we need for executing bets on these three selections from the oddsline, according to the criteria for the betting decision used in *Example 9-1*. The key to the above parameters are:

```
Time, Course, Name, Betfair market id, Betfair selection Id, Price Asked, Back (or lay),
Exchange (or SP), Persistence (or none), and BSP liability.
```

Leaving this file "as is" and running ***bet_execution.pl*** will attempt to execute bets at the market price, which is the price captured in the input file. A simple way to test ***bet_execution.pl***, however, without risking cash is to manually edit the *selections* file and change the market price parameters to "1000". Then, save the *selections* file and run the betting execution program which will pick it up, providing the directory names used are all correct.

Provided there are no errors in running the program itself (i.e. the program compiles correctly), successful execution will result in 3 unmatched bets at 1000. (hopefully it is not necessary to add that if risking capital is to be avoided, then the program should not be tested on a race where the selections are indeed available at decimal odds of 1000).

At this point we can go into the Betfair website interface and cancel the bets with no harm done to our cash balance. Or, better still, we can check that the bets have been placed using *Example 10-2* to retrieve unmatched bets from the command line, thus testing that this program works in its own right. Then, we can take the opportunity to test the "cancel unmatched bets" program by entering the **marketId** of the event and running *Example 10-4*.

At some point, provided that we want to go ahead and implement a particular strategy, we will want to test the program live, running end to end, prior to making any further

improvements. We can test live, again without risking any capital, by simulating bets to be placed and using a results database to assess what would have happened (we discuss this in Chapter 15). However, we will still require an effective live implementation even with a simulated betting approach, so if we have no confidence in the implementation itself, this is not a sound method of simulating a production run for a new strategy.

The prerequisite therefore, is establishing reliable implementation. We opt to do this in the next section, running our example strategy live to small stakes, having done unit tests on the framework beforehand. So in the next section we jump in with both feet for our first real taste of automation – then make further assessments on possible improvement afterwards.

Testing The Example Strategy

As far as this test is concerned, the most important thing we are looking for is that the mechanics of the strategy work as planned. At the same time, this is also the first "acid test" for automation, since we are out of the realms of theory and are risking capital for the first time.

Even though we are testing to show our implementation works exactly as we planned, a test such as this can still give immediate insights into improvements that can be made in the strategy itself. It is unrealistic to assume that all eventualities will have been covered at the planning stage, so any new phase of testing is generally useful to provide food for thought on how to improve. However, whatever the results in terms of profit or loss, we cannot realistically assess from one day if the oddsline itself is useful (at highlighting overlays by predicting realistic chances). Therefore, the question of the oddsline itself is a matter to be revisited and improved in a separate process, after we have gathered more data – which we discuss in Chapter 14. So, with those caveats aside, let's run through the test conditions.

Some readers may have noticed that data used for some previous examples has related to racing on October 13th 2007. The date has no particular significance, save for having been selected for running a live test, the one we are about to comment upon, using the basic oddsline example generated in Chapter 6, in conjunction with the other elements of our automated strategy.

The date falls at the end of the Flat season. The previous weekend has seen the running of the richest race in Europe at Longchamp, the *Arc De Triomphe*, (won in 2007 by *Dylan Thomas* after a lengthy stewards enquiry).

There are arguably other possible finales to the English Flat season, including Champions Day at Newmarket at the end of October, but whatever the case, the jumping season starts to take greater and greater precedence in the fixture list.

So it is with Saturday October 13th – we have some high class racing at Ascot, York's last meeting of the year, and more than 3 jumps fixtures entering the mix. If we look back to Chapter 5, *Betting Process 101*, where we automatically retrieved all events for this day, we can run a query to find all the UK meetings, listed as follows:

- Ascot
- York
- Bangor
- Hexham
- Chepstow
- Kempton

For our purposes, the quantity and variety of racing make for a useful day's testing since there are over 40 races at 6 meetings, including different code types, namely Flat turf (at Ascot and York); Flat All Weather (at Kempton), and Jumps (at Bangor, Hexham and Chepstow). Not only are there six meetings for the automated strategy to contend with, the strategy will consider every race as a possible betting medium, according to the rules of the betting decision program.

From an implementation point of view, the example strategy therefore has the widest possible applicability to any other strategy we might put together in future. In other words, if successful, we know that our programs can be adapted to dynamically schedule any strategy, to consider any number of races every day, according to any criteria we choose - without any manual intervention required on a day to day or race to race basis.

Rules for the example strategy

For the live test we are going to concentrate on the example used within the Automated Framework, namely finding and betting overlays according to an oddsline.

This example oddsline is based on data automatically parsed from the Racing Post using the method outlined in Chapter 6. The method of producing the example oddsline is hypothetical in the sense it is based on a third party view of what are the most useful factors for predicting outcomes, and upon what is an untested way of scoring these factors. On the other hand, the oddsline method is based on the Postdata table which uses different factors to rate horses depending on the race type and conditions, so it is a reasonable starting hypothesis to differentiate ratings for all 40 races, regardless of type.

However, the type of output produced in creating this oddsline is the important feature for our examples, being generic for any oddsline method. This output consists of a means of identifying each contender, together with a tissue price, based on an estimate of the probability of success in the event.

Below is a recap on the rules to be implemented for the strategy which uses the oddsline (N.B. these rules were also discussed within the context of example automation requirements in Chapter 7, and the setup for the programs that run them is described in detail there):

1. Take the oddsline, work out whether there are non-runners. If there are non-runners, make a reduction in the oddsline equivalent to the percentage takeout of the removed runners and re-adjust prices to a 100% book. So, we build into our program an algorithm for calculating that.

2. From the field that goes to post, i.e. our remaining oddsline, take the top third of the field by dividing the remaining field size by three (rounding down to the nearest integer), and compare each runner's market price to the oddsline price for that runner. This is our list of possible contenders to bet on.

Our logic here is only to look for overlays amongst the runners that are scored highest, (i.e. that are highly ranked), as opposed to looking for overlays in the whole oddsline. This is a recognition of the limitations of the method used, since to score highly, a horse must have proved its worthiness, and we are therefore betting on proven attributes. To score lowly, a horse may be very poor or may simply be an unknown quantity. Combined with the fact that the tissue prices – and hence overlays – become more suspect towards the bottom of the oddsline, we are adjusting in advance for likely weaknesses in the pricing method. Finally, we take the top third of runners as opposed to, say, a static number such as the first 5, so that we don't end up in the trap just described, for example, backing everything that is an overlay in a 5 horse race. The more backtesting and confidence in the oddsline prices at all levels, the more such strategies would make sense.

3. If the oddsline price is lower than the market price, (that is to we have spotted an overlay and our oddsline makes the percentage chance of winning higher than the market does), we save the runner details. If the oddsline price is higher than the market, technically this would indicate that the contender is overbet, and we might think about laying it. Since the Postdata oddsline is not thoroughly tested, we will instead leave it.

4. After we have assessed our runners against the market prices available and saved those that we wish to bet on, we write out all relevant details to a selections file, ready for betting execution. We ask, not for too much, but for the back price that we last took from the market. That way, given the time lapse in obtaining the price and a computer program executing, we are pretty sure to get on.

5. We execute the bet, and if successfully placed, retrieve the bet details and store them for later.

Summary of set up:

We have linked all aspects of the process to create and implement an automated system that bets untouched by human hand. It pulls data in the morning from the Racing Post and repurposes it, clobbering the previous day's file. It identifies the relevant market Id for each race from Betfair, the runners, or selection Ids from Betfair, and keeps count of which events and runners have already been dealt with.

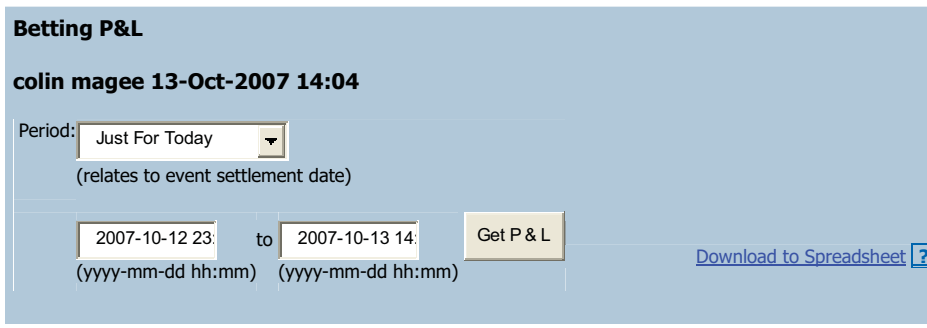
For each race, 5 minutes before the off, the automated strategy fetches Betfair prices, brings the oddsline up to date by taking care of non-runners, and then weighs up the chances generated by the oddsline versus those available in the market. And then it bets. And writes us a report, so we can see exactly what happened.

The programs themselves are being run on one of the oldest computers in the author's hardware set up, on a PC bought in 1998, OS Suse Linux 8.0, running in France, on a 2GB broadband connection. In terms of a robust hardware setup for automatically betting on the Betfair exchange, only a dial-up modem would make it less optimal.

And They're Off!

So to the first race. The programs in each betting phase are faithfully initiated by the (Linux) operating system, counting down the minutes until the next scheduled job in the background.

Let's have a look at P&L for the betting account used, by interrogating the website interface after the day's schedule has had a chance to get going, with the first few races over, at 2 pm.



Horse Racing: GBP101.00 Total P&L: GBP101.00

Horse Racing — Showing **1 - 3** of 3 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Ascot 13th Oct : 1m4f Grp 3	13-Oct-07 13:45	13-Oct-07 13:58	121.00
Horse Racing / Chep 13th Oct : 3m Hcap Chs	13-Oct-07 13:25	13-Oct-07 13:37	-10.00
Horse Racing / Ascot 13th Oct : 5f Grp 3	13-Oct-07 13:10	13-Oct-07 13:15	-10.00

Figure 12-1: Betfair P&L after first few races in live test

Over £100 in profit after 3 races – a good start! And now we encounter one of the many curiosities of automated betting– we are in profit (in this particular instance) but at this stage we have no idea why. Clearly the strategy has backed a winner, but what and where?

Staying with the standard Betfair interface for a moment, let's find out by drilling into the details of the profitable position, for the first race shown in *Figure 12-1*:

Horse Racing: GBP101.00 Total P&L: GBP101.00

Horse Racing > Ascot 13th Oct : 1m4f Grp 3 — Showing **1 - 2** of 2 Selections

Selection	Odds	Stake (GBP)	Bid type	Placed	Profit/loss (GBP)
Trick Or Treat	27.00	5.00	Back	13-Oct-07 13:40	130.00
Winter Sunrise	15.50	5.00	Back	13-Oct-07 13:40	-5.00
				Back subtotal:	125.00
*Average odds: On Off				Lay subtotal:	0.00
				Market subtotal:	125.00
				Commission @ 3.2%:	4.00
				Net Market Total:	**121.00**

Profit and Loss is shown net of commission.

All times are UKT [?] unless otherwise stated.

Figure 12-2: Betfair P&L for 13:45 Ascot

Was the winning bet on *Trick Or Treat* all matched in one bet? Let's see by drilling down again interactively.

Betting P&L

colin magee 13-Oct-2007 14:05

Period: Just For Today

(relates to event settlement date)

2007-10-12 23: to 2007-10-13 14: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet [?]

Horse Racing: GBP101.00 Total P&L: GBP101.00

Horse Racing > : 1m4f Grp 3 > Trick Or Treat — Showing **1 - 1** of 1 Bets

Selection	Odds	Stake (GBP)	Bid type	Placed	Matched (GMT)	Profit/loss (GBP)
Trick Or Treat	27	5.00	Back	13-Oct-07 13:40	13-Oct-07 13:40	130.00

Profit and Loss is shown net of commission.

All times are UKT [?] unless otherwise stated.

Figure 12-3: Betfair P&L for Trick or Treat

Yes: one £5 bet, matched at the first time of asking.

Of course, we could have produced all this information automatically using the API. But at this stage we are running a test, so checking back and forth between programs that are being tested and the live interface is a good thing.

Let's tie the results above together with the standard output produced from our betting programs themselves, by looking at the betting report that was generated by our programs for this race:

```
Race time = 1.45, Course = Ascot
------------------------------  Get prices for 1.45
20631295
Login Successful!
Successful call:
12 runner prices found
Race time = 1.45, Course = Ascot
1.45, Ascot, Brisk Breeze, 7.33, 3.95
1.45, Ascot, Trick Or Treat, 7.33, 27.0
1.45, Ascot, Queen's Best, 8.25, 4.0
1.45, Ascot, Winter Sunrise, 11.0, 15.5
1.45, Ascot, Kayah, 13.2, 13.0
1.45, Ascot, Satulagi, 13.2, 220.0
1.45, Ascot, Samira Gold, 13.2, 4.5
1.45, Ascot, Athenian Way, 16.5, 100.0
1.45, Ascot, Rising Cross, 16.5, 32.0
1.45, Ascot, Loulwa, 16.5, 75.0
1.45, Ascot, Dash To The Front, 16.5, 34.0
1.45, Ascot, Party, 22.0, 30.0
ALL RUN
Trick Or Treat, 20631295, 1244199, 27.0, B, 5.00
Winter Sunrise, 20631295, 2429313, 15.5, B, 5.00


------------------------------  Bet horses
Sat Oct 13 13:40:04 BST 2007
Login Successful!
Following bets placed:
Ascot, 1.45,  Trick Or Treat, B, 27.0, 3815255965
Following bets placed:
Ascot, 1.45,  Winter Sunrise, B, 15.5, 3815256033
Bet horses finished for 1.45
```

Figure 12-4: Automated Program Report for 1:45 Ascot

The program report is particular to our strategy and gives us much more detail on the decision making process, so that we not only see *which* bets were placed, but *why*.

Recall that the program report includes elements produced as text output from each program that makes up the automated framework. These are executed prior to each race automatically, by the executable file ***bet_oddsline_strategy***.

This file consists of the following programs:

```
/home/aeb/get_market_prices.pl;
/home/aeb/bet_formation.pl;
/home/aeb/execute_bets.pl;
```

Each program contributes to building up the overall report, concatenating its output to the existing file. We can make this output as detailed or as short as we wish. The example report provides a basic overview of the whole process. At this stage we are mainly concerned with capturing output that can show us if there are any problems and checking that the strategy is working as it was planned.

Thus ***get_market_prices.pl*** has some lines added from our original example in Chapter 8 to kick off the process, printing out the time of the race, course, and output from the calls made, including the market Id below:

```
Race time = 1.45, Course = Ascot
----------------------------  Get prices for 1.45
20631295
Login Successful!
Successful call:
12 runner prices found
```

get_market_prices.pl writes all market prices to a temporary file, *inter_prices*, which is subsequently picked up by ***bet_formation.pl***. This example program, covered in *Example 9-1*, takes in the prebuilt oddsline itself, and the prices from the file, then works out overlays according to our betting strategy. By way of output, we first verify that we have picked up the right race time and course details again,

```
Race time = 1.45, Course = Ascot
```

Then we print out the oddsline, with the time, course, horse name, oddsline tissue price, and Betfair back price, as follows:

```
1.45, Ascot, Brisk Breeze, 7.33, 3.95
1.45, Ascot, Trick Or Treat, 7.33, 27.0
1.45, Ascot, Queen's Best, 8.25, 4.0
1.45, Ascot, Winter Sunrise, 11.0, 15.5
1.45, Ascot, Kayah, 13.2, 13.0
1.45, Ascot, Satulagi, 13.2, 220.0
1.45, Ascot, Samira Gold, 13.2, 4.5
1.45, Ascot, Athenian Way, 16.5, 100.0
1.45, Ascot, Rising Cross, 16.5, 32.0
1.45, Ascot, Loulwa, 16.5, 75.0
1.45, Ascot, Dash To The Front, 16.5, 34.0
1.45, Ascot, Party, 22.0, 30.0
```

If all runner names are found, the program prints out “ALL RUN” or “NON RUNNERS” so that we can quickly verify whether that is indeed the case, so in the 1.45 Ascot :

```
ALL RUN
```

Now we can start to see *why* we backed Trick or Treat.

The oddsline strategy was set by us to automatically choose the top third of the field ranked by the oddsline and seek overlays, so *Brisk Breeze, Trick or Treat, Queen’s Best and Winter Sunrise* should all have been evaluated. According to the tissue used in this oddsline, Brisk Breeze was overbet at 3.95, Trick or Treat overlaid, Queen’s Best overbet at 4.0 and Winter Sunrise overlaid. Thus, the program writes out 2 sets of horse details to the ***selections*** file, which will subsequently be bet, as follows:

```
Trick Or Treat, 20631295, 1244199, 27.0, B, 5.00
Winter Sunrise, 20631295, 2429313, 15.5, B, 5.00
```

Finally, our last program in the sequence, ***execute_bets.pl***, as covered in Chapter 10, *Example 10-1* picks up the bet details from the `selections` file and places the bets. The final part of the output from the program report is produced by this program, which lets us know that we are into the betting execution stage, and the system time when the program was run, as in:

```
------------------------------  Bet horses
Sat Oct 13 13:40:04 BST 2007
```

Then we report that the program connected ok with the Betfair API

```
Login Successful!
```

Lastly, if bets were written out to the selections file, we report that those bets have been successfully placed by the program, which we can verify through the presence of the bet identification number as the last part field in the line.

```
Following bets placed:
Ascot, 1.45,  Trick Or Treat, B, 27.0, 3815255965
Following bets placed:
Ascot, 1.45,  Winter Sunrise, B, 15.5, 3815256033
Bet horses finished for 1.45
```

Having covered the meaning of the program output, we retire from monitoring the automatic betting program – even though it is a test, it would be a shame not to profit from the point of letting the programs run automatically. So let's return to the scene an hour later, and test the output from the programs versus the Betfair interface.

So, an hour or so later, we can look at the state of the automatic betting account just after the 15:45 at York.

Betting P&L

colin magee 13-Oct-2007 15:53

Period: Just For Today

(relates to event settlement date)

2007-10-12 23: to 2007-10-13 15: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet ?

Horse Racing: -GBP3.50 Total P&L: -GBP3.50

Horse Racing — Showing **1 - 18** of 18 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / York 13th Oct : 6f Hcap	13-Oct-07 15:45	13-Oct-07 15:51	-20.00
Horse Racing / Chep 13th Oct : 2m Nov Hrd	13-Oct-07 15:40	13-Oct-07 15:48	-15.00
Horse Racing / Ascot 13th Oct : 7f Cond Stks	13-Oct-07 15:35	13-Oct-07 15:44	-10.00
Horse Racing / Bang 13th Oct : 3m Hcap Chs	13-Oct-07 15:25	13-Oct-07 15:36	60.50

Horse Racing / Hex 13th Oct : 3m1f Class Chs	13-Oct-07 15:15	13-Oct-07 15:24	-10.00
Horse Racing / York 13th Oct : 6f Listed	13-Oct-07 15:10	13-Oct-07 15:16	-15.00
Horse Racing / Chep 13th Oct : 2m4f Hcap Hrd	13-Oct-07 15:05	13-Oct-07 15:13	-15.00
Horse Racing / Ascot 13th Oct : 1m Grp 3	13-Oct-07 15:00	13-Oct-07 15:04	-5.00
Horse Racing / York 13th Oct : 2m2f Hcap	13-Oct-07 14:40	13-Oct-07 14:47	-5.00
Horse Racing / Chep 13th Oct : 3m Nov Chs	13-Oct-07 14:35	13-Oct-07 14:43	-10.00
Horse Racing / Bang 13th Oct : 2m1f Hcap Hrd	13-Oct-07 14:25	13-Oct-07 14:34	-10.00
Horse Racing / Ascot 13th Oct : 1m4f Hcap	13-Oct-07 14:20	13-Oct-07 14:30	-10.00
Horse Racing / Hex 13th Oct : 2m Beg Chs	13-Oct-07 14:15	13-Oct-07 14:21	-10.00
Horse Racing / York 13th Oct : 1m1f Hcap	13-Oct-07 14:10	13-Oct-07 14:18	-15.00
Horse Racing / Chep 13th Oct : 2m Hcap Hrd	13-Oct-07 14:00	13-Oct-07 14:05	-15.00
Horse Racing / Ascot 13th Oct : 1m4f Grp 3	13-Oct-07 13:45	13-Oct-07 13:58	121.00
Horse Racing / Chep 13th Oct : 3m Hcap Chs	13-Oct-07 13:25	13-Oct-07 13:37	-10.00
Horse Racing / Ascot 13th Oct : 5f Grp 3	13-Oct-07 13:10	13-Oct-07 13:15	-10.00

Profit and Loss is shown net of commission.

All times are UKT [?] unless otherwise stated.

Figure 12-5: Betfair P&L after 3.45 York

Not so good. Looks like the automatic profits previously made have been wiped out. In fact, theoretical P&L at Betfair prices (i.e. minus commission), would show that the account or betting strategy is marginally in front, but the -£3.50 reflects the actual profit and loss with commission deducted – commission running around 3.2% in this particular account at this time.

Time to leave it be and come back after the day's Turf racing is done, with only the Kempton All Weather remaining.

Well, here is the position after the "proper" day's racing:

Betting P&L

colin magee 13-Oct-2007 17:57

Period: Just For Today

(relates to event settlement date)

2007-10-12 23: to 2007-10-13 17: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet [?]

Horse Racing: GBP63.32 Total P&L: GBP63.32

Horse Racing Showing **1 - 20** of 31 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Hex 13th Oct : 2m NHF	13-Oct-07 17:30	13-Oct-07 17:50	65.34

Horse Racing / York 13th Oct : 1m6f Hcap	13-Oct-07 17:20	13-Oct-07 17:43	-25.00
Horse Racing / Bang 13th Oct : 2m1f Hcap Chs	13-Oct-07 17:10	13-Oct-07 17:19	-5.00
Horse Racing / Hex 13th Oct : 2m4f Hcap Chs	13-Oct-07 17:00	13-Oct-07 17:10	-10.00
Horse Racing / York 13th Oct : 1m2f Hcap	13-Oct-07 16:50	13-Oct-07 16:56	33.88
Horse Racing / Chep 13th Oct : 2m NHF	13-Oct-07 16:45	13-Oct-07 16:53	-15.00
Horse Racing / Ascot 13th Oct : 5f Hcap	13-Oct-07 16:40	13-Oct-07 16:49	-10.00
Horse Racing / Bang 13th Oct : 2m1f Nov Hrd	13-Oct-07 16:35	13-Oct-07 16:46	-5.00
Horse Racing / Hex 13th Oct : 2m4f Mdn Hrd	13-Oct-07 16:25	13-Oct-07 16:34	-20.00
Horse Racing / York 13th Oct : 1m Mdn Stks	13-Oct-07 16:20	13-Oct-07 16:26	-10.00
Horse Racing / Chep 13th Oct : 2m Nov Hrd	13-Oct-07 16:15	13-Oct-07 16:23	38.72
Horse Racing / Ascot 13th Oct : 1m2f Hcap	13-Oct-07 16:10	13-Oct-07 16:14	-5.00
Horse Racing / Hex 13th Oct : 3m Hcap Hrd	13-Oct-07 15:50	13-Oct-07 15:58	33.88
Horse Racing / York 13th Oct : 6f Hcap	13-Oct-07 15:45	13-Oct-07 15:51	-20.00
Horse Racing / Chep 13th Oct : 2m Nov Hrd	13-Oct-07 15:40	13-Oct-07 15:48	-15.00
Horse Racing / Ascot 13th Oct : 7f Cond Stks	13-Oct-07 15:35	13-Oct-07 15:44	-10.00
Horse Racing / Bang 13th Oct : 3m Hcap Chs	13-Oct-07 15:25	13-Oct-07 15:36	60.50
Horse Racing / Hex 13th Oct : 3m1f Class Chs	13-Oct-07 15:15	13-Oct-07 15:24	-10.00
Horse Racing / York 13th Oct : 6f Listed	13-Oct-07 15:10	13-Oct-07 15:16	-15.00
Horse Racing / Chep 13th Oct : 2m4f Hcap Hrd	13-Oct-07 15:05	13-Oct-07 15:13	-15.00

Pages: 1 2 of 2 Pages

Jump to page:

Profit and Loss is shown net of commission.

All times are UKT ? Unless otherwise stated.

Figure 12-6: Betfair P&L before Evening Racing

Marvellous - we recouped the position.
£63 up. Or over 12 points (since we are betting to £5 level stakes), after commission has been deducted.

This profit for the day could still be eaten by activities at Kempton, of course. We could simply kill the process at this point, but that would not do our cause of testing a full day's racing for all meetings any good. In any case, regarding potential downside, we shouldn't lose much if anything from this position, with 6 races to go and 12 points in hand. Let's come back at the end of the day and find out.

Results summary

Below the final screenshot for betting P&L, taken less than an hour after the last race is over at Kempton. The first thing to notice is that the live test does indeed seem to have gone through the card automatically.

Betting P&L

colin magee 13-Oct-2007 22:27

Period: Just For Today

(relates to event settlement date)

2007-10-12 23: to 2007-10-13 22: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet ?

Horse Racing: GBP155.37 Total P&L: GBP155.37

Horse Racing Showing **1 - 20** of 38 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Kemp 13th Oct : 1m Hcap	13-Oct-07 21:20	13-Oct-07 21:26	-5.00
Horse Racing / Kemp 13th Oct : 1m4f Hcap	13-Oct-07 20:50	13-Oct-07 21:00	-5.00
Horse Racing / Kemp 13th Oct : 6f Mdn Stks	13-Oct-07 20:20	13-Oct-07 20:25	-5.00
Horse Racing / Kemp 13th Oct : 7f Mdn Stks	13-Oct-07 19:50	13-Oct-07 20:02	55.81
Horse Racing / Kemp 13th Oct : 1m2f Claim Stks	13-Oct-07 19:20	13-Oct-07 19:28	28.07
Horse Racing / Kemp 13th Oct : 5f Hcap	13-Oct-07 18:50	13-Oct-07 18:56	-5.00
Horse Racing / Bang 13th Oct : 2m1f NHF	13-Oct-07 17:40	13-Oct-07 18:01	28.17
Horse Racing / Hex 13th Oct : 2m NHF	13-Oct-07 17:30	13-Oct-07 17:50	65.34
Horse Racing / York 13th Oct : 1m6f Hcap	13-Oct-07 17:20	13-Oct-07 17:43	-25.00
Horse Racing / Bang 13th Oct : 2m1f Hcap Chs	13-Oct-07 17:10	13-Oct-07 17:19	-5.00
Horse Racing / Hex 13th Oct : 2m4f Hcap Chs	13-Oct-07 17:00	13-Oct-07 17:10	-10.00
Horse Racing / York 13th Oct : 1m2f Hcap	13-Oct-07 16:50	13-Oct-07 16:56	33.88
Horse Racing / Chep 13th Oct : 2m NHF	13-Oct-07 16:45	13-Oct-07 16:53	-15.00
Horse Racing / Ascot 13th Oct : 5f Hcap	13-Oct-07 16:40	13-Oct-07 16:49	-10.00
Horse Racing / Bang 13th Oct : 2m1f Nov Hrd	13-Oct-07 16:35	13-Oct-07 16:46	-5.00
Horse Racing / Hex 13th Oct : 2m4f Mdn Hrd	13-Oct-07 16:25	13-Oct-07 16:34	-20.00
Horse Racing / York 13th Oct : 1m Mdn Stks	13-Oct-07 16:20	13-Oct-07 16:26	-10.00
Horse Racing / Chep 13th Oct : 2m Nov Hrd	13-Oct-07 16:15	13-Oct-07 16:23	38.72
Horse Racing / Ascot 13th Oct : 1m2f Hcap	13-Oct-07 16:10	13-Oct-07 16:14	-5.00
Horse Racing / Hex 13th Oct : 3m Hcap Hrd	13-Oct-07 15:50	13-Oct-07 15:58	33.88

Pages: 1 2 of 2 Pages

Figure 12-7: Final P&L for UK Racing Saturday 13th October

Furthermore, the last 6 races at Kempton have indeed pushed us further ahead. We end the day a very satisfactory £155.37 ahead, after commissions have been deducted. Or, 35 points profit to level stakes.

These are not the shabbiest results for any strategy, although as we said at the start of the test, the main objective is to test the process of automation for this strategy on a live basis rather than to assess the oddsline explicitly. A large number of bets were placed, but further testing over a period of time would tell us much more. We accumulate more data on the performance of the oddsline and compare it to another oddsline in Chapter 14.

First, however, we need to come back to the implementation. On the face of it, the strategy was correctly implemented, and we made a profit. It achieved the objective of complete automation in that no manual intervention was necessary on the day – the oddsline was generated automatically, the schedule was generated automatically, and all programs executed on time, as planned.

However, only by using the detailed program report and checking against other sources can we verify this assumption. We also need to review whether the implementation, in addition to being correct, is easy to maintain and to modify, and make decisions about whether it should be changed prior to implementing a strategy on a permanent basis.

We therefore assess and discuss improvements for the strategy on this basis in the next Chapter.

Chapter 13: Assessing and Improving the Implementation

In Chapter 12 we ran a live test based upon integrating all the elements of the example framework for an example oddsline strategy, for all the races on Saturday 13th October 2007.

We know that the implementation was a qualified success, since the oddsline was generated automatically and the schedule executed automatically for every race on the day.

However, we cannot be sure that the strategy was implemented correctly for every race without reviewing more detailed output. Additionally, within a final review of a test prior to implementing any strategy live, we will also give consideration to cleaning up the code and making it easier to maintain and run.

Assessing Output from the Live Test

In Chapter 12, *Figure 12-4* we listed and then described the content of the program report for the 1.45 Ascot, explaining how the report was put together and the elements within it.

The work required to assess whether or not the strategy has run correctly is to review the program report for all races.

The report runs chronologically from the first race in the sequence - the 1.10 at Ascot – to the last – the 9.20 at Kempton.

In no particular order, the most important things we are looking to check in terms of the test are:

- **Automatic scheduling** – all races have been captured and the strategy is executed at the right time before each race (i.e. 5 minutes before the off)
- **Oddsline pickup** – a valid oddsline is available for each race assessed by the strategy
- **Race identification** – all races have been correctly identified and matched between the oddsline and Betfair data sources
- **Runner identification** – all runners have been correctly identified and matched between the oddsline and Betfair data sources
- **Betfair calls** have worked correctly – in the event of the calls failing, we need to identify whether that is due to a hardware, software or a Betfair problem.
- **Strategy rules** – where the above has been correctly implemented, that the strategy is implemented exactly as we had planned

Here we will reprint the report only for identified anomalies since the full program report from the live test on 13th October 2007 runs to some 10 pages. However, the full report is also available online at *www.betwise.co.uk* if readers want to review that first – there are some clear issues that can be identified by "eyeballing" this.

We can quickly verify that all races picked up a valid oddsline and that all races were executed in sequence – scanning down the list of races tells us so. We can also find that the operating system log will tell us which programs were executed automatically and at what time, so this can be used to verify that each program ran at the allotted time, 5 minutes before the race in question. However, this cannot be seen from the test report. It would be more convenient to see this data for each race, also.

This is our first improvement for the live system. Further, we currently have no differentiation between the system time and the Betfair time. They are assumed to be identical whereas in fact there can be discrepancies. We will therefore prefer the Betfair time and capture it for the race in question, to verify that we are betting on the race within an acceptable time range.

Next we consider the question of runner identification and Betfair calls.

A quick and easy method for identifying problems with both is to use the "NON_RUNNER" identification within each race. Our program logic dictates that whenever a name in the oddsline is not associated with a name and a back price in the hash %inter_prices, we print "NON-RUNNER" to the report and then revalue the tissue prices to accommodate the non-runner.

Therefore, what we are looking for in the program report is to verify that all horses printed with this label were in fact non-runners. If we have a label NON-RUNNER when the horse was running, this means a back price was not found for that runner – either because of problems with identifying the runner name and matching it to a back price or because no prices exist.

Looking at the report and tallying this with the results for the day, we can identify a few instances of horses that ran but were incorrectly identified as non-runners. Within *Figure 13-1*, we can see that *Tonic Du Charmil* in the 1.25 at Chepstow and then *I'll Do It Today* in the 3.50 at Hexham were both identified as non-runners but in fact ran in the race.

```
##############################  NEW TRADE  ################################
Race time = 1.25, Course = Chepstow
------------------------------  Get prices for 1.25
20633172
Login Successful!
Successful call:
13 runner prices found
Race time = 1.25, Course = Chepstow
1.25, Chepstow, Tribal Venture, 8.50, 7.2
1.25, Chepstow, Champagne Harry, 8.50, 10.0
1.25, Chepstow, Tonic Du Charmil, 11.3, NON_RUNNER
1.25, Chepstow, Sultan Fontenaille, 11.3, 19.0
1.25, Chepstow, Natal, 13.6, 4.4
1.25, Chepstow, Tom Sayers, 13.6, 9.8
1.25, Chepstow, Bowleaze, 13.6, 8.8
1.25, Chepstow, Two Miles West, 13.6, 11.0
```

```
1.25, Chepstow, Gan Eagla, 13.6, 14.5
1.25, Chepstow, Tanterari, 13.6, 24.0
1.25, Chepstow, Le Duc, 17.0, 24.0
1.25, Chepstow, Celtic Boy, 22.6, 65.0
1.25, Chepstow, Unleash, 22.6, 230.0
NON_RUNNERS
Champagne Harry, 20633172, 245346, 10.0, B, 5.00
Sultan Fontenaille, 20633172, 1190611, 19.0, B, 5.00


------------------------------  Bet horses
Sat Oct 13 13:20:04 BST 2007
Login Successful!
Following bets placed:
Chepstow, 1.25,  Champagne Harry, B, 10.0, 3815098564
Following bets placed:
Chepstow, 1.25,  Sultan Fontenaille, B, 19.0, 3815098618
Bet horses finished for 1.25


############################  NEW TRADE  ###############################
Race time = 3.50, Course = Hexham
------------------------------  Get prices for 3.50
20633197
Login Successful!
Successful call:
14 runner prices found
Race time = 3.50, Course = Hexham
3.50, Hexham, Ballyboe Boy, 8.30, 12.5
3.50, Hexham, Willie The Fish, 10.3, 10.0
3.50, Hexham, Court One, 10.3, 26.0
3.50, Hexham, Talarive, 11.8, 9.6
3.50, Hexham, Colourful Life, 11.8, 10.5
3.50, Hexham, Greenfort Brave, 11.8, 50.0
3.50, Hexham, Sadler's Cove, 11.8, 20.0
3.50, Hexham, I'll Do It Today, 16.6, NON_RUNNER
3.50, Hexham, Mulligan's Pride, 16.6, 12.5
3.50, Hexham, Caesar's Palace, 16.6, 44.0
3.50, Hexham, Willywont He, 20.7, 5.7
3.50, Hexham, Essifer, 20.7, 12.0
3.50, Hexham, Ramblees Holly, 20.7, NON_RUNNER
3.50, Hexham, Spartan Warrior, 83.0, 12.5
3.50, Hexham, Longstone Lass, 83.0, 250.0
NON_RUNNERS
Ballyboe Boy, 20633197, 377958, 12.5, B, 5.00
Willie The Fish, 20633197, 675962, 10.0, B, 5.00
Court One, 20633197, 40674, 26.0, B, 5.00


------------------------------  Bet horses
Sat Oct 13 15:45:05 BST 2007
Login Successful!
Following bets placed:
Hexham, 3.50,  Ballyboe Boy, B, 12.5, 3816747745
Following bets placed:
Hexham, 3.50,  Willie The Fish, B, 10.0, 3816747882
Following bets placed:
Hexham, 3.50,  Court One, B, 26.0, 3816748025
Bet horses finished for 3.50
```

Figure 13-1: Races with instances of incorrectly identified non-runners

So, what do *Tonic Du Charmil* in the 1.25 at Chepstow and then *I'll Do It Today* in the 3.50 at Hexham have in common? Why were these names not recognized?

Firstly, the names of the horses in the report, printed from the oddsline, are the correct Weatherby registered names, so any search should in theory have matched the runner names.

We searched for the correct names in Betfair, so what did Betfair store as the names for these horses?

Tonic du Charmil in the 1.25 at Chepstow
I'LI Do It Today in the 3.50 at Hexham

We recognized in the example program for the framework in Chapter 9 that some transformation was needed between full horse names and the names used on Betfair, since the Betfair convention is to drop any apostrophes. Thus, in *Example 9 -1*, we first drop apostrophes from the oddsline name, then use this to match up against the Betfair hash %inter_prices, where names are the key, in order to retrieve a back price, as below:

```
$horse_bfname =~ s/'//g;
my $horse_bfname_odds = @ {$inter_prices{$horse_bfname} }[0];
```

However, this clearly does not allow any room for error. The system *should* work but doesn't. This doesn't just happen with Betfair names. No two different data sources can be assumed to be the same. So, we can well imagine a situation where the names in our oddsline contain some error or typo, and do match with a correct version (minus apostrophes) in Betfair.

For our example errors, we can see that in the case of *Tonic Du Charmil*, there was a capitalisation problem with *Du* and in the case of *I'll Do It Today* an erroneous capital letter introduced in the second letter.

It turns out that capitalisation problems, apostrophes, hyphens and other punctuation problems are the worst enemies for matching names from different sources. Rarely if ever are horses missing letters in their names between any two reputable sources and apart from introducing fuzzy matching techniques, there is not much more that can be done to legislate for such errors. But clearly we need to make a further improvement to eliminate any matching problems due to the punctuation and capitalisation issues mentioned. So we will ensure that when the name is compared, it is compared as lower case throughout, and condense it to remove all apostrophes, hyphens, and spaces:

```
my $short_horse_name = lc($horse);   #reduce name to lower case throughout
$short_horse_name =~ s/\'|-|\s+//g;  #remove apostrophes, hyphens and spaces
```

Let's now take a look at the final piece of anomalous output from the program report (continuing from *Figure 13-1*), as shown in *Figure 13-2*:

```
############################  NEW TRADE  ##############################
Race time = 2.55, Course = Bangor
----------------------------  Get prices for 2.55
20634386
Race time = 2.55, Course = Bangor
2.55, Bangor, Magical Quest, 5.44, NON_RUNNER
2.55, Bangor, Cyd Charisse, 8.17, NON_RUNNER
2.55, Bangor, Ethan's Star, 8.17, NON_RUNNER
2.55, Bangor, So Many Questions, 8.17, NON_RUNNER
```

```
2.55, Bangor, Time Out, 12.2, NON_RUNNER
2.55, Bangor, Oxford De Lagarde, 16.3, NON_RUNNER
2.55, Bangor, Dream Garden, 16.3, NON_RUNNER
2.55, Bangor, Satan's Sister, 16.3, NON_RUNNER
2.55, Bangor, Souffleur, 24.5, NON_RUNNER
2.55, Bangor, Bendarshaan, 24.5, NON_RUNNER
2.55, Bangor, Clear The Way, 24.5, NON_RUNNER
2.55, Bangor, Go On George, 24.5, NON_RUNNER
2.55, Bangor, Gumlayloy, 49.0, NON_RUNNER
NON_RUNNERS

------------------------------  Bet horses
Sat Oct 13 14:50:13 BST 2007
Login Successful!
"Selections" file is not written or blank

Sat Oct 13 14:50:14 BST 2007
```

Figure 13-2: Instance of a failed call

In the case of the 2.55 at Bangor, every runner in the race is declared a non-runner; or rather, no price has been found for any of the runners and the report output simply states “Non runner”.

However, we can see from the absence of the “Login successful” message in the first part of the report output (the part that deals with retrieving prices) that there has been failure to connect with the Betfair API. An interrogation of the system log confirms this also – so we have missed assessing and potentially betting on the whole race. What went wrong?

Well, we deliberately used the clunkiest hardware and infrastructure set up in the pack for this test – an old machine and an internet connection to the Betfair Exchange from the French countryside - and this is the consequence. The error is due to a timeout error – it can also be due to a failure to connect due to inconsistent connectivity. I know from past experience that this type of error is not unusual on this machine for this setup and frequency of calls to the API (the failure rate being approximately one call in every 80 – 100). In the case of timeout errors we can make some changes to improve this, although there would remain a root problem with the setup.

In any case, the exception would prove the rule, since we can say that even with a clunky hardware the strategy is more than 95% successfully implemented.

Nonetheless, we will later seek to improve upon the implementation by running on a much faster set up, with hardware based in the UK, avoiding timeout errors and also connectivity problems.

In addition to the issues with connectivity that we have identified, it is also worth noting that there is occasionally Betfair API downtime. However, again the exception proves the general rule (of reliability in automation), since the API itself has an exceptionally good uptime record – 99.99% plus.

Apart from the issues identified above, there is no problem with the strategy rules. These have been implemented correctly, albeit in the case of non-runners that they may have considered one horse too many in the rankings, preferring the next horse in the list

as opposed to the incorrectly identified non-runner, or missed out on a winner in some cases.

Therefore, let's look at what impact these implementation errors have had upon the success or otherwise of the strategy. In the case of the non-runners, we have identified a few – *Tonic Du Charmil* would in fact have been an overlay in the 1.25 Chepstow (since it was within the top third of the field), which would have meant one extra loser. *I'll Do It Today* in the 3.50 at Hexham was outside the top third ranked in the field, so even though its price was missed and it was erroneously declared a non-runner, this fact makes absolutely no difference to the implementation of this particular strategy.

What about the entirely missed race, the 2.55 at Bangor, what would have happened there if things had worked out correctly?

Well, *Souffleur*, who was off the radar for the oddsline, won the race. *Magical Quest*, *Cyd Charisse* and *So Many Questions* would all have been overlays if the program had picked up the Betfair prices, so effectively the fact that the program didn't work saved a 3 point loss. On the other hand, we could equally have missed a 100/1 winner. If the strategy is a good one, our interest is in executing correctly as close to 100% as possible.

However, it is worth noting that the fact the errors identified are an almost entirely random effect. For example, even if one race in 100 is missed, provided it is removed completely at random, then a strategy which has proven it is profitable over all 100 races will still be profitable in the long run.

Improving the Implementation

We spotted the following problems with the live test, as discussed in the previous section:

- Timeout with internet connection responsible for one missed race
- Insufficient consideration to matching horses' names, responsible for a few mismatches and missed or incorrect selections.

We also have resolution for both problems for our final implementation.

The first issue is resolved by running a faster, more reliable connection to avoid dropped connections and timeouts (if this were not possible, we could also address the timeout issue independently).

The second issue requires a little more work on the programming side. It is resolved by implementing a better matching algorithm for horse names from different data sources (in the example case, between the Racing Post and Betfair). However, this should be comprehensive enough to ensure matching for any data sources.

In practice, we must therefore *replace* the lines of code below which are used to match the oddsline horse name within *Example 9-1*:

```
$horse_bfname =~ s/'//g;
my $horse_bfname_odds = @ {$inter_prices{$horse_bfname} }[0];
```

Instead they are to be substituted with the following *correct* code

```
my $short_horse_name = lc($horse);   #reduce name to lower case throughout
$short_horse_name =~ s/\'|-|\s+//g;  #remove apostrophes, hyphens and spaces
```

However, this is only one half of the solution. We now have the problem that the transformed oddsline name (all in lower case with no punctuation and no spaces), will not match the unaltered Betfair name. We did not have this issue in *Example 9-1* due to our assumption that we were fixing the oddsline to match the Betfair convention and that the latter would always be consistently implemented.

Thus, prior to matching the horse from the oddsline, we also have to transform the Betfair names. We can do this by creating an array of Betfair names from our **`%inter_prices`** hash (where all the keys are Betfair names), and setting up a foreach loop to read in the list of Betfair names and transforming the name to the same shortened form as the oddsline name (minus capitals, spaces and punctuation). Then we create a new hash with the transformed Betfair name as the key, and the actual Betfair name (that can be used to look up any value in `%inter_prices`) as the value.

```
#create array of Betfair horse names that make up all runners in the race

my @bf_names = keys(%inter_prices)

#transform the Betfair horse name to eliminate any potential anomalies
#this will now be the same as the transformed oddsline name
#store that in a new hash as the key, with the full Betfair name as the value

foreach my $name (@bf_names) {
        #print "$name\n";
my $short_bf_name = lc($name);
$short_bf_name =~ s/\'|-|\s+//g;

#create a new hash so the oddsline "short name" can be matched to the actual Betfair name

$bf_short_names{$short_bf_name} = $name;

};
```

At this point the oddsline short name that we have created should be the same as the Betfair short name and can therefore be used to look up the actual Betfair horse name - which in turn can be used to retrieve the back price of that runner.

```
#now use the oddsline short name to look up the actual Betfair name
$horse_bfname = $bf_short_names{$short_horse_name};

#we can now continue as per the original Example 9-1
my $horse_bfname_odds = @ {$inter_prices{$horse_bfname} }[0];
```

Now we have corrected for the matching problem and *Example 9-1* will work correctly within the context of this strategy.

So, we have improved the individual programs used in the live test.

However, there are wider issues to be addressed before we press the "Go" button on the current implementation – namely combining all programs into one.

Simply taking the case of adapting *Example 9-1*, it would be easier to first store runner prices by runner ID and then create a Perl hash whereby the runner Id can be looked up by using the transformed horse name (same as oddsline), as below:

```
#create a new hash so the oddsline "short name" can be used to look up Betfair Ids

$runnerId = $static_runner_data{$name};
$bf_short_names{$short_bf_name} = $runnerId;
```

For this to work, we would first need to have all the runner prices in memory, accessible by runner ID rather than name. This would mean altering our fixed file hash, **`%inter_prices`** (which is generated by the previous ***`get_market_prices.pl`*** program) in order to do that. If all runner prices (to all levels) were available within memory within the context of the program in *Example 9-1*, as they were within the context of *Example 8-2*, when the `inter_prices` file was created, then it would be easy to create whatever data structure we wanted.

The problem with the "live test" implementation is that we built the strategy up from the building blocks (i.e. individual programs) of the automated framework, adapted for an oddsline example, and simply pasted them together within an executable file. Thus, at the time of betting, we have 3 programs that run sequentially, executed by the command ***`bet_oddsline_strategy`***, as follows:

```
#replace all paths with your own
/home/aeb/get_market_prices.pl;
/home/aeb/bet_formation.pl;
/home/aeb/execute_bets.pl;
```

Both ***`get_market_prices.pl`*** and ***`execute_bets.pl`*** double up on the process of opening the libraries and modules to be used in common with both programs, whilst ***`bet_formation.pl`*** also has overhead loading files and interacting with the operating system. In all programs, the suitability of writing out files and then reading them in within the context of the next program is questionable. This occurs most notably for writing out all market prices when they are already available for use within a data structure in `get_market_prices.pl`, then reading in a file and recreating that data structure in a subsequent program. Again, we write out the selections for betting to a file within `bet_formation.pl`, and then read in that file within `execute_bets.pl`. We could simply use the data structures that are already there and significantly increase efficiency, so that is what we will do.

Amalgamating all the programs also enables us to maintain a single production version of the program, in far fewer lines of code. We can eliminate all the "top and tail" code within the programs (subsequent to `get_market_prices.pl`) since we have already loaded all the prerequisite items.

Thus we can start to combine the code from *Example 8-2*, *Example 9-1* and *Example 10-1* together to eliminate redundant elements and create a single, more efficient program.

Thus the repeated module loading in `execute_bets.pl` can disappear since it has already been done within the context of *Example 8-2*:

```
#       prerequisite modules
use lib "/home/aeb/lib";
use BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use DBI;
use strict;
```

Likewise declaration of login variables and the call to the ***login*** function can also be executed once, eliminating the following lines of code (in programs subsequent to *get_market_prices.pl*):

```
#       login variables
my $username = "username";
my $password = "password";
my $productId = "82";           #Free Access API access code


#       login to the Betfair API
my %login = login($username, $password, $productId);
my $token = $login{sessionToken};
my $login_error = $login{errorCode};
```

As can duplicated error catching – we only need to log in once.

```
if ( !($login_error =~ /OK/) )
 {
   print "Failed login:\n";
   print "$login_error";
 }
 else
 {
        print "Login Successful!\n";
 }
```

We can also make a reference to our events schedule only once, and read in events only once. We can rewrite the schedule in preparation for the next race, rather than burying this feature within one of the programs and worrying unduly about the order of execution of the 3 different programs. Now there is one program, the correct event details will be used throughout, and upon exit the schedule will be reset for the next iteration of the single program:

```
#       Read in the next event and write out events schedule ready for the next event
open (EVENTS, "< /home/aeb/events_schedule");
open (TEMP, "> /home/aeb/events_temp");
my @events = <EVENTS>;
my $event_details = shift(@events);
foreach my $details (@events) { print TEMP "$details";}
close (EVENTS);
close (TEMP);
rename("/home/aeb/events_temp", "/home/aeb/events_schedule");

$event_details =~ s/^\s+|\s+//g;
print "$event_details\n";
my ($race_time, $race_course, $marketId) = split(/,/, $event_details);
```

Wherever files are opened for receiving details - such as market prices and betting selections - we can simply read prices and selections straight from data structures in memory.

The program or betting report also becomes much easier to write, since it is no longer split over the actions of 3 programs, but visible within one program.

We can now take the opportunity to capture the market timestamp from Betfair and to capture any error codes whenever a Betfair call is made. This will all make for a fuller report than we showed in the first live test.

Many examples of the new reporting format are shown in the Chapter 14, *Oddsline Strategies in Production*, under the section *Automatic Betting Diary.*

We can choose to use even more copious reporting, or less – that will be up to the reader in their adaptation of code for production. All the elements needed to do this are discussed within the context of this section, adapted from the original framework examples in Part 2.

Now we can save this program with an appropriate name – `oddsline_overlays.pl` for example, and replace the 3 programs previously named in `bet_oddsline_strategy` with this one, to execute 5 minutes before every race, as dictated by the rules in the dynamic scheduling program, *Example 7-2*.

Conclusion

We corrected some errors that were spotted in our assessment of the live test and created a single program to bet dynamically, using the oddsline, prior to the race. Creating a single final version of the program enables us to:

- Use data structures upon creation, without writing to, and reading from files
- Eliminate redundant code and speed up processing
- Produce more robust reporting with fuller information
- Make the program generic for any oddsline we might throw at it

We are not claiming this is a perfect implementation. But it is getting better and better as a result of the testing process. At a certain point, where everything works as we expect, and the output suits, we have reached a sufficient standard or bar where we can “go live”.

We now have a generic program for dynamically betting oddslines. This means that it can be easily adapted to back any oddsline, with any combination of rules, since the process of generating the oddsline is separate from the program that dynamically backs the oddsline.

All we require is that the format used for the oddsline can be read into the dynamic betting program and that we can split out per line: the race in question, the runner and the tissue price. Since we have separated the process of betting from the assessment method used, any assessment method for contenders leading to an oddsline can be used.

The process of creating the events schedule is also separate from the dynamic betting program. The execution program simply reads the events that are presented to it. For any other strategy, a different schedule can be used.

Chapter 14: Oddsline Strategies in Production

We now have a suite of programs to automatically bet any oddsline without manual intervention, from setting up scheduling to execution. So, provided the oddsline itself can be automatically generated or automatically acquired, any betting strategy which incorporates an oddsline can be fully automated.

In this chapter we will test this further by running more than one oddsline for the same race. At the same time, a longer trial running two oddslines in parallel will enable us to show what the business of running automated strategies is really like in practice.

To begin, we need to look further afield for an oddsline than the example we generated in Chapter 6, which was based around the Racing Post Postdata table, although we will also maintain this one for comparison purposes.

We are looking for an oddsline method which is based on a sound method, tested over a period of time and which, given the right approach and knowledge any bettor could generate on an automated basis. In other words, the oddsline used should be based on quantitative methods applied to data which is readily available (as opposed to privileged information which is qualitative and inaccessible to some, such as stable gossip, and cannot be easily automated).

The ratings available from *www.equiformratings.co.uk* fit the bill in that they are based on a consistent mathematical method, which itself uses standard data elements also available from the *SmartForm Racing Database*, at *www.betwise.co.uk*.

The ratings are automatically generated by a daily program and converted to tissue prices, creating an oddsline which is available as a proprietary service. For the production run, we automatically download the prepared oddsline, although we cover the background behind the method in the next section.

Thereafter we recap on the conditions and objectives for the production run, and then discuss the business end of the production run itself.

Background on Oddslines used

For the production run we are using the improved program discussed in the previous chapter, together with our automatic scheduling programs, to bet two different oddslines with the same rules on exactly the same races.

Postdata oddsline

The methodology for our first oddsline will come as no surprise, being the oddsline used in the live test and put together in the example framework, specifically in Chapter 6, for automatically repurposing data from the Racing Post website.

Although the oddsline is based purely on assumptions regarding the significance and weighting of the factors in the Racing Post Postdata, as opposed to testing the resulting rankings against results, it has shown a small profit during our live test.

However, it should be expected that this oddsline will require the most work in future and may produce volatile results if used without backtesting the accuracy of the oddsline predictions against the market. This is the way we will be using the oddsline i.e. sight unseen, during our production run.

At the same time, without testing the oddsline, we cannot say what improvement to the oddsline is needed. Ideally we would do this outside the context of a live environment of course, building up the data over some time and testing it. On the other hand, there are few strategies that are put into automation that do not carry some risk, even if backtested, so it is real and will serve as a useful comparison point with an alternative. We can also confirm that the framework now works exactly as we expect it to following the improvements covered in the previous chapter.

Subsequent to the production run, we will also analyse the accuracy of the Postdata oddsline using the results that we have gathered, in order to gain insight into the process of assessing and improving an oddsline itself.

Equiformratings oddsline

The Equiform oddsline is the professional addition to the oddsline stable for the production run. The ratings and tissue prices can be acquired directly from *www.equiformratings.co.uk*.

Although the exact method for generating the Equiform ratings is proprietary, and we cannot provide the code, we can provide some background. Also, since the method uses standard data which is also available in the *Smartform Racing Database* (which is itself automatically updated) similar quantitative methods for generating oddslines can be created and automatically implemented by other bettors for themselves.

At the core of the methodology behind the Equiformratings oddsline is the belief that past performances of a horse can be expressed in terms of weight. The past performances are called past merit ratings (PMRs). The PMRs are used to estimate the likely performance of each horse in each race for which selections are made. These estimates are called expected merit ratings (EMRs).

The weight to be carried is then subtracted from the EMRs and the answer obtained is used as the criterion for judging the most likely winner, next most likely winner and so on. The difference between EMR and weight is called the rating criterion (the RC).

For the purposes of our oddsline test, the ratings for all races in the UK will be considered, although analysis of past results shows that some race types are more successful for top ranked selections than others.

The PMRs are of course achieved under a variety of race conditions, different courses, goings, different racing codes and sometimes even different racing jurisdictions.

Estimating EMRs from PMRs is tricky but necessary. The method used is based on a huge number of tests carried out on UK and Irish racing results from 1998 to 2004, some 66,000 races in all. Factors taken into account are:

1. draw bias
2. weight for age (WFA)
3. weight for experience (WFE)
4. time since last run
5. recency of earlier runs compared to most recent run
6. race distance of past races compared to today's race distance
7. codes of past runs (meaning AW, turf, hurdle, chase)
8. gender
9. headgear
10. jockey
11. trainer (only used for debutantes)

The WFA tables have been determined without any reference to official tables, using a "what works best" approach.

WFE tables were introduced to give, for example, recognition to the fact that a physiologically mature horse that switches from the flat to the jumps will improve in performance for a few years notwithstanding the fact that the horse may already have reached the peak of its physical development. They are also useful in judging the chances, for example, of a horse that switches from turf to AW or the other way round. Draw bias is estimated over the 5 calendar years preceding the calendar year of selection. Both PMR's as well as EMR's are adjusted for draw bias.

Jockey skills are measured by comparing the actual performance of riders' mounts with the expected performance ignoring the jockey. Trainer skills are measured in a way similar to the way jockey skills are measured, after allowing for jockey. Finally, the RC's are used to estimate win probabilities, which are converted to "fair prices" and expressed in odds to 1 terms. These prices make up a 100% book.

For the purposes of our test, thanks to the proprietor of Equiformratings, we have automatically captured the ratings and tissue prices from a web page each day and parsed them to conform to the same format as the oddsline generated by example in Chapter 6. At the same time we convert the starting prices from the tissue to decimal odds by adding 1, and end up with a file that looks like and can be used within the framework in exactly the same way as the Postdata oddsline file we saw earlier, *formatted_oddsline*.

Production run rules

For both oddslines we will use exactly the same betting rules as implemented for the Postdata based oddsline during the live test.

This means taking the top third of the field after allowing for any non-runners, and betting any contender which is an overlay. Full criteria for implementing the strategy are covered in Chapter 12 under the subsection *Rules for the example strategy*.

We will follow both strategies for 2 weeks from the 18th November 2007 to 1st December 2007 inclusive. The strategies will cover all races in the UK during this time period, so that there can be a like for like comparison between oddslines. There is nothing special about the dates chosen, they simply fell in line with preparation of this part of the book.

For all bets we take whatever the current back price is 5 minutes before the race and immediately bet £5 per overlay.

As a real example of running automated strategies, we will leave the strategy running for both oddslines as they stand for the first week, whilst closely observing both strategies, as discussed in the introduction to the *Automatic Betting Diary* below. As we are risking real capital on two untried strategies, we will also review the implementation of the strategy after a week.

Whilst this would generally be an unnecessary step for any well tested strategy, and may be for the Equiformratings oddsline, for which there is plentiful backtesting, it is prudent to monitor and adjust the completely untested Postdata oddsline. Here we have no idea what to expect in the long run. In Chapter 15, we discuss "turning off" betting execution for the Postdata oddsline altogether and simply accumulate data so as to have sufficient data to test, assess and improve the Postdata oddsline without risking capital. However, turning off betting execution at this point does not create a genuine point of comparison with our other oddsline, and is not as useful for describing what can go wrong as well as right with a live strategy – however, it can be a further improvement to the process of creating new strategies, so we also cover that in the next Chapter.

Automatic Betting Diary

Whilst betting automatically, we want to make sure we know inside out how any automated strategy works in practice. This is necessary to be confident that we can leave the strategy alone with no real surprises. Of course, we will never remove risk, but at least we can familiarize ourselves with the operation of the strategy so as to be as comfortable with leaving it alone as we would be if we were betting each race live.

It means more than observing that the implementation conforms to the rules we specified (especially since we checked that for the live test), but also that the rules themselves are as useful in practice as they may have seemed in theory. Particularly in the early days of any strategy, it is important to get a "feel" for the way any strategy operates, to analyse any anomalies, and so on.

Two weeks is way too short a time frame for any perspective on long term profitability of a strategy, but our objective is rather to demonstrate the mechanisms involved in reviewing it, and give a sense of what happens in running any betting strategy on an automated basis - warts and all.

Hopefully this perspective can be useful in showing what type of results, together with their associated ups and downs, can be expected for those bettors who automate their betting for the first time. Of course, the exact results will vary by strategy type so each bettor should consider the expected behaviour of their own strategy first and foremost.

For the purposes of our production run, both strategies are betting into the same Betfair account. Aside from having sub-accounts set up (which is possible for accounts above a certain transaction volume), this is a useful way to reduce the overall commission payable to Betfair, otherwise each strategy would be subject to its own commission payment. By the same token, we also want to be able to monitor the profit and loss of each strategy separately, so we employ the methods discussed in Chapter 11, *Automated Record Keeping,* and write the results of each strategy out to a separate database table at the end of each day.

The diary is an abridged version of notes kept over the two week period. Since a number of observations and explanations are in common with subsequent days' betting, more detail has been left in the first few days' notes.

Week 1

18th – 24th November 2007

Everything – being the programs discussed in Part 2, with improvements to the betting program made in Part 3 - has been finalised and scheduled to execute automatically as **cron** jobs. Each **cron** job creates a dynamic schedule for each oddsline, each day. There is nothing to be done in terms of manual intervention during the first week of the diary but observe (or, for future reference if the racing does not appeal or, in case of more pressing priorities, to be doing something different).

However, in the course of observation, we do capture a few screen shots from the manual interface. For the first few days below, we also cover the processes used for monitoring the strategies and their performance in general – this information is common to all days, of course. Sample output from both oddslines, showing program and betting reports for all oddslines produced on the Sunday of the production run, is available at *www.betwise.co.uk*

Sunday 18th November

On the first day, being a Sunday with other things to do, we look at activity after racing has finished.

Let's start with looking at the type of information at our disposal, which will be common to all other days for the diary.

Firstly, we review the overall profit and loss situation for both strategies after commissions, using the website interface for the account balance as our point of departure (shown for familiarity, subsequently we show data gathered automatically).

Betting P&L

colin magee 18-Nov-2007 22:34

Period: Just For Today

(relates to event settlement date)

2007-11-18 00: to 2007-11-18 22: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet ?

Horse Racing: GBP42.72 Total P&L: GBP42.72

Horse Racing Showing **1 - 19** of 19 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Carl 18th Nov : 2m1f NHF	18-Nov-07 15:50	18-Nov-07 15:59	-20.00
Horse Racing / Font 18th Nov : 2m2f Hcap Chs	18-Nov-07 15:40	18-Nov-07 15:49	12.51
Horse Racing / Chelt 18th Nov : 2m5f Nov Hrd	18-Nov-07 15:30	18-Nov-07 15:38	-15.00
Horse Racing / Carl 18th Nov : 2m4f Hcap Chs	18-Nov-07 15:20	18-Nov-07 15:27	28.86
Horse Racing / Font 18th Nov : 3m4f Hcap Chs	18-Nov-07 15:10	18-Nov-07 15:21	-25.00
Horse Racing / Chelt 18th Nov : 2m Hcap Hrd	18-Nov-07 14:55	18-Nov-07 15:03	-40.00
Horse Racing / Carl 18th Nov : 2m1f Nov Hrd	18-Nov-07 14:45	18-Nov-07 14:54	-15.00
Horse Racing / Font 18th Nov : 2m2f Hcap Hrd	18-Nov-07 14:35	18-Nov-07 14:43	29.82
Horse Racing / Chelt 18th Nov : 2m Hcap Chs	18-Nov-07 14:20	18-Nov-07 14:28	-5.00
Horse Racing / Carl 18th Nov : 3m2f Hcap Chs	18-Nov-07 14:10	18-Nov-07 14:19	79.36
Horse Racing / Font 18th Nov : 2m2f Claim Hrd	18-Nov-07 14:00	18-Nov-07 14:06	-10.00
Horse Racing / Chelt 18th Nov : 2m5f Hcap Hrd	18-Nov-07 13:45	18-Nov-07 13:54	67.34
Horse Racing / Carl 18th Nov : 2m1f Hcap Hrd	18-Nov-07 13:35	18-Nov-07 13:42	-20.00
Horse Racing / Font 18th Nov : 2m6f Hcap Hrd	18-Nov-07 13:25	18-Nov-07 13:33	-20.00
Horse Racing / Chelt 18th Nov : 2m Nov Chs	18-Nov-07 13:10	18-Nov-07 13:17	-10.00
Horse Racing / Carl 18th Nov : 2m4f Nov Chs	18-Nov-07 13:00	18-Nov-07 13:07	-20.00
Horse Racing / Font 18th Nov : 2m2f Beg Chs	18-Nov-07 12:50	18-Nov-07 12:57	3.08
Horse Racing / Carl 18th Nov : 2m4f Nov Hrd	18-Nov-07 12:30	18-Nov-07 12:38	31.75
Horse Racing / Font 18th Nov : 2m4f Mdn Hrd	18-Nov-07 12:20	18-Nov-07 12:28	-10.00

Profit and Loss is shown net of commission.

All times are UKT ? unless otherwise stated.

Figure 14-1: Overall P&L November 18th 2007

The good news is that the first day results in profit, for £42.72 after commissions. However, this doesn't tell us anything about the performance of our individual strategies.

For that we look to the output from our end of day record keeping process, as discussed in Chapter 11, which collects all betting information according to strategy, based on captured bet Ids through the day, then writes all daily bet details for each strategy to the database.

Subsequently, a program can be scheduled to execute an automated database query to retrieve from each strategy record the total number of bets for the day, the total number of wins, and the P&L for the day, as shown in *Table 14-1*. In this data, we are interested in the raw performance of the strategy and therefore report results without commissions (since these vary by account and prevailing commission rate), although the commission data is also available for extraction, as discussed in Chapter 11.

For each strategy, displayed in the final column, we also calculate cumulative P&L for that strategy - for the first day, this is of course the same as actual P&L.

Table 14-1: P&L by Oddsline Strategy, November 18th 2007

Oddsline	Date	Bets	Wins	P&L	cum.P&L
Equiform	18/11/2007	34	4	71	71
Postdata	18/11/2007	33	3	-18.3	-18.3

This tells us what we need to know at a high level – the Equiform oddsline was positive in terms of profit, and the Postdata oddsline produced a loss. However, there are as many questions as answers. There were almost the same number of bets and winners for each strategy, so we do not know how or why each strategy produced its results.

To answer these questions, we can fall back on our improved program output for all detail at the lowest level, produced for every single race in the UK on the day. This shows the output from matching each oddsline to the available live prices 5 minutes before each race, as well as the overlays determined by the strategy within the top third of the field.

There are separate program reports from each strategy, which in the case of our examples were running from within different directories on the same server. Each program report for one of the key handicap races on the first day are reproduced in *Figure 14-2*, for the Postdata oddsline, and in *Figure 14-3* for the Equiform oddsline.

```
################################  NEW ODDSLINE  ################################

Cheltenham, 1.45
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       13:40:04.897
1.45, Cheltenham, Khasab, 13.70, 4.7
1.45, Cheltenham, Im Spartacus, 13.70, NON_RUNNER
1.45, Cheltenham, Leading Contender, 13.70, 12.5
1.45, Cheltenham, Hot Port, 15.22, 7.2
1.45, Cheltenham, Rajeh, 15.22, NON_RUNNER
1.45, Cheltenham, Vale Of Avocia, 17.12, 30.0
1.45, Cheltenham, Rustarix, 17.12, NON_RUNNER
```

```
1.45, Cheltenham, Portland Bill, 19.57, 25.0
1.45, Cheltenham, Dancing Lyra, 19.57, NON_RUNNER
1.45, Cheltenham, Counting House, 19.57, 24.0
1.45, Cheltenham, Rio De Janeiro, 19.57, 16.0
1.45, Cheltenham, Pocket Too, 22.83, 120.0
1.45, Cheltenham, Sobers, 22.83, 11.0
1.45, Cheltenham, The Sliotar, 27.40, 12.0
1.45, Cheltenham, Mendo, 34.25, 22.0
1.45, Cheltenham, Hereditary, 34.25, 160.0
1.45, Cheltenham, La Dame Brune, 34.25, 29.0
1.45, Cheltenham, Paradi, 45.67, 18.5
1.45, Cheltenham, Go Free, 45.67, 140.0
1.45, Cheltenham, At The Money, 45.67, 18.0
1.45, Cheltenham, Court Ruler, 68.50, 95.0
1.45, Cheltenham, Boss Imperial, 68.50, 210.0
1.45, Cheltenham, Robin De Sherwood, 68.50, 40.0
1.45, Cheltenham, Top The Charts, 137.00, NON_RUNNER
NON_RUNNERS

Bet top third overlays:
Selection:   1.45, Cheltenham, Leading Contender, 20694871, 1277440
Bet placed: 4011675011, B, £5.00,  12.5
Selection:   1.45, Cheltenham, Vale Of Avocia, 20694871, 1442574
Bet placed: 4011675052, B, £5.00,  30.0
Selection:   1.45, Cheltenham, Portland Bill, 20694871, 1344282
Bet placed: 4011675113, B, £5.00,  25.0
Selection:   1.45, Cheltenham, Counting House, 20694871, 1225820
Bet placed: 4011675137, B, £5.00,  24.0

Oddsline strategy finished at systime: Sun Nov 18 13:40:06 GMT 2007
```

Figure 14-2: Postdata oddsline report, Cheltenham 1:45, 18/11/2007

```
################################  NEW ODDSLINE  ###################################

Cheltenham, 1:45
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       13:40:04.519
1:45, Cheltenham, Leading Contender, 8.06, 12.5
1:45, Cheltenham, Im Spartacus, 9.62, NON_RUNNER
1:45, Cheltenham, Khasab, 11.36, 4.7
1:45, Cheltenham, Portland Bill, 12.05, 25.0
1:45, Cheltenham, Rustarix, 14.93, NON_RUNNER
1:45, Cheltenham, Hot Port, 16.95, 7.2
1:45, Cheltenham, Rio De Janeiro, 17.86, 16.0
1:45, Cheltenham, Mendo, 20, 22.0
1:45, Cheltenham, Counting House, 20.41, 24.0
1:45, Cheltenham, Sobers, 21.74, 11.0
1:45, Cheltenham, Rajeh, 21.74, NON_RUNNER
1:45, Cheltenham, The Sliotar, 23.26, 12.0
1:45, Cheltenham, Top The Charts, 28.57, NON_RUNNER
1:45, Cheltenham, Dancing Lyra, 29.41, NON_RUNNER
1:45, Cheltenham, At The Money, 40, 18.0
1:45, Cheltenham, Paradi, 43.48, 18.5
1:45, Cheltenham, Vale Of Avocia, 66.67, 30.0
1:45, Cheltenham, Court Ruler, 111.11, 95.0
1:45, Cheltenham, Pocket Too, 111.11, 120.0
1:45, Cheltenham, Boss Imperial, 111.11, 210.0
1:45, Cheltenham, Go Free, 125, 140.0
1:45, Cheltenham, La Dame Brune, 125, 29.0
1:45, Cheltenham, Robin De Sherwood, 166.67, 40.0
1:45, Cheltenham, Hereditary, 250, 160.0
NON_RUNNERS

Bet top third overlays:
```

```
Selection:   1:45, Cheltenham, Leading Contender, 20694871, 1277440
Bet placed: 4011674893, B, £5.00,  12.5
Selection:   1:45, Cheltenham, Portland Bill, 20694871, 1344282
Bet placed: 4011674940, B, £5.00,  25.0
Selection:   1:45, Cheltenham, Rio de Janeiro, 20694871, 837310
Bet placed: 4011675012, B, £5.00,  16.0
Selection:   1:45, Cheltenham, Mendo, 20694871, 1071645
Bet placed: 4011675048, B, £5.00,  22.0

Oddsline strategy finished at systime: Sun Nov 18 13:40:05 GMT 2007
```

Figure 14-3: Equiform oddsline report, Cheltenham 1:45, 18/11/2007

Not only was this one of the better races of the day, but this handicap at Cheltenham was also pivotal to the difference in performance on the day between both strategies. The finishing order for the 1:45 at Cheltenham was as follows:

```
1. Mendo              16/1
2. Vale of Avocia     18/1
3. Sobers             9/1
4. Leading Contender  8/1
```

Mendo as the winner generated £105 prior to commissions for the Equiform oddsline, without which the strategy would have resulted in a loss. Although the selection processes for both oddslines had selected the 4th horse, *Leading Contender* as well as the unplaced *Portland Bill,* it was *Mendo* at a back price of 22.0 which was a unique selection only for the Equiform oddsline. Ironically, the second horse, *Vale of Avocia,* which was beaten only 1 length and in contention according to the race comments, was a unique selection for Postdata. If *Vale of Avocia* had won at even higher odds than *Mendo,* of 30.0, it would have completely reversed the P&L position between oddslines strategies (since by the same token *Mendo* would have lost), as well as producing a greater overall profit in the account on the day.

Such is the fine line between winning and losing in terms of any individual race. It also shows us that the overall P&L position on the day eventually boils down to one race. Although it is too early to come to any conclusions, it is clear that if the balance of results stays like this, throwing up an average of 1 winner in every 8 or 9 bets, then the success of both oddslines depends on identifying a good smattering of long priced winners to make up for the majority of losing bets. In this case the honours went to the Equiform oddsline.

Monday 19th November 2007

It is moderate fayre today with regard to the quality of horses competing, with meetings at Wolverhampton, Kempton and Leicester.

Yesterday we caught up on results at the end of the day, today we will follow progress intermittently during the course of racing.

It seems that the first few races produce the automated equivalent of going two steps forwards and two steps backwards, so that aggregated profits from both oddslines have put us £15 up by the 2.10 at Leicester. This is enough of a capital reserve to pay for a

mere 3 bets before returning to a zero position, which, whilst we are considering the top third of the field, can easily be surpassed by one strategy alone within one race.

Sure enough, there are 3 bets placed in total by both oddslines in the very next race, the 2.10 at Leicester. Below we show only the Equiform oddsline, however the Postdata oddsline strategy produced one overlay within the top 3 runners, also *Yo Pedro* at decimal odds of 6.6.

```
################################  NEW ODDSLINE  ################################

Leicester, 2:10
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       14:05:01.445
2:10, Leicester, Airman, 4.41, NON_RUNNER
2:10, Leicester, L'Oiseau De Feu, 6.13, NON_RUNNER
2:10, Leicester, Baarrij, 6.76, NON_RUNNER
2:10, Leicester, Devilfishpoker Com, 6.94, 6.0
2:10, Leicester, Yo Pedro, 8.77, 6.6
2:10, Leicester, Salvestro, 12.2, 9.4
2:10, Leicester, Floodlight Fantasy, 21.28, 25.0
2:10, Leicester, Mujamead, 25, 2.1
2:10, Leicester, Riverweld, 50, 22.0
2:10, Leicester, Miss Bustino, 66.67, 40.0
2:10, Leicester, Yankey, 251, 120.0
NON_RUNNERS

Bet top third overlays:
Selection:   2:10, Leicester, Devilfishpoker Com, 20696369, 1464356
Bet placed: 4016436296, B, £5.00,  6.0
Selection:   2:10, Leicester, Yo Pedro, 20696369, 1036997
Bet placed: 4016436321, B, £5.00,  6.6

Oddsline strategy finished at systime: Mon Nov 19 14:05:03 GMT 2007
```

Figure 14-4: Equiform oddsline report, Leicester 2:10, 19/11/2007

Sure enough, 3 bets are indeed placed by the combination of both oddslines, for 2 different runners. (We note in passing that the top 3 originally rated in the race are all non-runners – as if the quality of racing wasn't already poor, the highest ranked horses are also dropping out!) Losers for both would be sufficient to wipe out our meagre profit at this stage in the day.

Following the race live, we have a clear interest in *Yo Pedro* winning since it is carrying two bets. However, this one is not in contention. *Devilfishpoker.com*, on the other hand is, and has a good chance to win it when *Mujamead,* the favourite and race leader, blunders around in the latter stages of the race and is picked up off the floor by Tony McCoy. However, *Devilfishpoker.com* shows reluctance to take advantage of any error and the champion jockey proves yet again why he is a champion. So our profit is wiped out. Back to level pegging on the day.

The day continues in much the same vein with moderate racing and a slight edge maintained by the operation of both oddsline strategies, albeit for a large number of bets. Nevertheless, the overall P&L position forges slowly ahead so that the profit before the last race of the day at Kempton stands at £40.52.

This time, combining the selections of both oddslines, there are 4 different overlays found at the top of the oddslines, with *Recalcitrant* being a common overlay. Taking account of the overall position of bets in the race, the following outcomes are the ones which interest us: *Fantasy Ride* wins £22, *Recalcitrant* £81.50, and *Blocklow* over £100. However, our biggest winner would be *Sir Sandicliffe* at £155. With a total of 5 bets placed by both oddslines, a win for any other runner will lose us £25.

Blocklow is in the lead for most of the race, tracked by *Fantasy Ride*. *Highest Esteem*, the favourite and looking easily the best horse, comes from off the pace to lead and stay on well, with *Fantasy Ride* second best. So, it's £25 down on the last race. Still, on the bright side that's still £15 profit on the second day, all automatic. What conclusions can we draw?

Let's run our profit programs, starting with an interactive capture of the screen for the Betfair account which both strategies are using

Betting P&L

Colin magee 19-Nov-2007 19:45

Period: Just For Today

(relates to event settlement date)

2007-11-19 00: to 2007-11-19 19: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet ?

Horse Racing: GBP15.52 Total P&L: GBP15.52

Horse Racing Showing **1 - 20** of 20 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Kemp 19th Nov : 1m4f Hcap	19-Nov-07 16:20	19-Nov-07 16:28	-25.00
Horse Racing / Kemp 19th Nov : 6f Hcap	19-Nov-07 15:50	19-Nov-07 16:00	-15.00
Horse Racing / Leic 19th Nov : 2m Nov Hrd	19-Nov-07 15:40	19-Nov-07 15:51	3.36
Horse Racing / Wolv 19th Nov : 6f Mdn Stks	19-Nov-07 15:30	19-Nov-07 15:36	-25.00
Horse Racing / Kemp 19th Nov : 7f Hcap	19-Nov-07 15:20	19-Nov-07 15:33	36.05
Horse Racing / Leic 19th Nov : 2m4f Hcap Hrd	19-Nov-07 15:10	19-Nov-07 15:20	22.58
Horse Racing / Wolv 19th Nov : 5f Hcap	19-Nov-07 15:00	19-Nov-07 15:04	1.92
Horse Racing / Kemp 19th Nov : 1m Mdn Stks	19-Nov-07 14:50	19-Nov-07 14:58	36.04
Horse Racing / Leic 19th Nov : 2m Hcap Hrd	19-Nov-07 14:40	19-Nov-07 14:49	-20.00
Horse Racing / Wolv 19th Nov : 2m Hcap	19-Nov-07 14:30	19-Nov-07 14:37	-15.00
Horse Racing / Kemp 19th Nov : 1m2f Hcap	19-Nov-07 14:20	19-Nov-07 14:30	15.38
Horse Racing / Leic 19th Nov : 2m Sell Hrd	19-Nov-07 14:10	19-Nov-07 14:20	-15.00
Horse Racing / Wolv 19th Nov : 1m Hcap	19-Nov-07 14:00	19-Nov-07 14:03	6.87
Horse Racing / Kemp 19th Nov : 1m2f Hcap	19-Nov-07 13:50	19-Nov-07 13:57	38.44
Horse Racing / Leic 19th Nov : 2m Nov Hrd	19-Nov-07 13:40	19-Nov-07 13:49	-10.00

Horse Racing / Wolv 19th Nov : 7f Sell Stks	19-Nov-07 13:30	19-Nov-07 13:33	-30.00
Horse Racing / Kemp 19th Nov : 1m2f Claim Stks	19-Nov-07 13:20	19-Nov-07 13:26	-20.00
Horse Racing / Leic 19th Nov : 2m4f Hcap Hrd	19-Nov-07 13:10	19-Nov-07 13:20	26.91
Horse Racing / Wolv 19th Nov : 6f Claim Stks	19-Nov-07 13:00	19-Nov-07 13:06	-10.00
Horse Racing / Kemp 19th Nov : 5f Class Stks	19-Nov-07 12:50	19-Nov-07 12:54	12.97

Profit and Loss is shown net of commission.

All times are UKT [?] unless otherwise stated.

Figure 14-5: Account P&L for racing on Monday 19th September 2007

So, £15.52 ahead after all commissions have been deducted (for measuring the P&L of the strategies, we do so without commission, since this will depend on each user's own account).

For following all the ups and downs of the racing during the day, that would be a poor wage indeed. Fortunately, it is not necessary to do so when betting automatically.

Having followed the racing interactively for both strategies, we can see a clear contrast with many traditional betting strategies. If playing interactively, it would be a time consuming exercise indeed to consider every race in the day and identify overlays in each, allowing for non-runners, then to back them all with the same betting criteria without any error in execution.

Not all automated strategies create such a volume of bets (although many create more), but the ability to do so reliably, in order to press home any marginal edge over the market, as well as the freedom to do other things throughout the day's racing, is one of the key advantages that can be gained from automation.

By the same token, in a strategy which yields a high number of bets with the intention of creating a long term advantage, we cannot expect to see much progress by looking at any individual bet in the strategy. So far, we have picked out key races that tell us something about the way the strategy is operating.

So, in operating an automated strategy, patience is a virtue. If the profitability of a strategy depends on a certain volume of bets, we cannot look at an individual race in the same way as if, for example, we had identified an unexposed horse with great potential and waited a few weeks until its next engagement. One race for an automated strategy based on assessing all overlays against an oddsline throughout the day is simply a single data point in a large number.

So, although profits are creeping up only marginally, they are still creeping up – *overall*, although it is clear that all oddslines strategies are not created equal – in *Table 14-2* we can see a summary of how each strategy is faring at the end of the second day:

Table 14-2: P&L by Oddsline Strategy, November 19th 2007

Oddsline	Date	Bets	Wins	P&L	cum.P&L
Equiform	19/11/2007	43	8	100.16	171.16
Postdata	19/11/2007	33	2	-76.5	-94.8

One strategy is accumulating profits, the other losses.

If this production run were not for demonstration purposes (and without the cushion of the Equiform oddsline) we should be concerned about operating the Postdata oddsline alone. Even though it is only the second day, knowing that this strategy is based on the untested oddsline means we cannot have any confidence that it will right itself in the long run, and we need to keep a close eye on it.

Tuesday 20th November

Another moderate day's racing today.

One of the early results is worthy of further analysis.

Sovereign King hacks up in the 1:10 at Folkestone by 4 lengths. With the Alan King stable establishing itself as one of the top yards in National Hunt and being in fine fettle over the previous 2 weeks it's a surprise that a relatively unexposed runner in a moderate race is allowed to go off at a starting price of 66/1. Even the supposedly more "fancied" of his two runners in the race starts at 25/1 - and duly comes second. The traditional layers' straight forecast pays £1026.08 to a £1 stake (we'll hope the stable staff had a couple of quid on).

From the point of view of our automated strategy, whilst things are tighter with traditional bookmakers at the top of the market, contenders at long starting prices such as *Sovereign King* are often factors higher on the exchanges, and this one is no exception, as shown in the Postdata oddsline in *Figure 14-6* below.

```
################################  NEW ODDSLINE  ################################

Folkestone, 1.10
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       13:05:01.288
1.10, Folkestone, Matcho Pierji, 4.44, 4.2
1.10, Folkestone, Master Medic, 5.71, 7.8
1.10, Folkestone, Kid Charlemagne, 6.67, 2.24
1.10, Folkestone, Uncle Eli, 6.67, 22.0
1.10, Folkestone, Sovereign King, 13.33, 160.0
1.10, Folkestone, Regal Quote, 20.00, 13.5
1.10, Folkestone, Champion De Sou, 20.00, 490.0
1.10, Folkestone, Darksideofthemoon, 20.00, 22.0
1.10, Folkestone, Oh Crick, 20.00, 38.0
1.10, Folkestone, Chancery Lad, 40.00, 140.0
ALL RUN

Bet top third overlays:
Selection:   1.10, Folkestone, Master Medic, 20697290, 2610451
```

```
Bet placed: 4020089232, B, £5.00,  7.8

Oddsline strategy finished at systime: Tue Nov 20 13:05:01 GMT 2007
```

Figure 14-6: Postdata oddsline report, Folkestone 1:10, 20/11/2007

The oddsline method rates all the runners in a tight band, and *Sovereign King* is an overlay by a factor of around 14 times at 160.0, to tissue odds of 13.33. However, *Sovereign King* is only 5th ranked overall, and, due to our field size and the rules of the strategy in place, we consider the top 3 only. So we miss out on the long priced winner, but no doubt we also miss out on a number of losers for the same reasons. Again, without the benefit of testing the Postdata oddsline thoroughly before playing it, we cannot know whether to laugh or cry about this.

Otherwise, it's a quiet day for the oddsline strategies in general, mainly due to small fields (comprising many novice and non-handicap events) and strong favourites. Between the oddslines, however, there is some difference in both the bets identified and the performance. For the Postdata oddsline, it is generally rarer that a favourite is priced under the market price (and therefore indicating an overlay) in small fields due to the tighter scoring range from the method used. Even if the winner is ranked highest, the odds produced by the scoring method tend to be too high to produce an overlay. For the Equiform oddsline, that is not the case, and the pricing tends to be more accurate.

As far as the type of overlays identified by the Postdata oddsline are concerned, a case in point occurs a few races later on the Folkestone card in the 2.40. This time the race is a more hotly contested handicap than in the jumps meeting at Fakenham, with a bigger field, where, as a result, the market prices are more evenly distributed. The prices for the top contenders produced by our Postdata scores are similarly likely to fall under some of the market prices on offer, thereby producing more bets for the higher ranked runners.

```
##################################  NEW ODDSLINE  ##################################

Folkestone, 2.40
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       14:35:03.543
2.40, Folkestone, Inaro, 8.75, 17.0
2.40, Folkestone, Fieldsofclover, 10.00, 11.0
2.40, Folkestone, Cossack Dancer, 10.00, 38.0
2.40, Folkestone, John Diamond, 10.00, NON_RUNNER
2.40, Folkestone, Black De Bessy, 10.00, 8.4
2.40, Folkestone, Berengario, 10.00, 15.0
2.40, Folkestone, Herecomestanley, 10.00, 10.0
2.40, Folkestone, Minuit De Cotte, 11.67, 4.5
2.40, Folkestone, The Hardy Boy, 14.00, 14.5
2.40, Folkestone, Coach Lane, 17.50, 14.0
2.40, Folkestone, Bagan, 17.50, 5.6
2.40, Folkestone, Barton Flower, 70.00, 38.0
NON_RUNNERS

Bet top third overlays:
Selection:   2.40, Folkestone, Inaro, 20697296, 982780
Bet placed: 4020790371, B, £5.00,  17.0
Selection:   2.40, Folkestone, Fieldsofclover, 20697296, 170725
Bet placed: 4020790383, B, £5.00,  11.0
Selection:   2.40, Folkestone, Cossack Dancer, 20697296, 389646
```

```
Bet placed: 4020790403, B, £5.00,  38.0

Oddsline strategy finished at systime: Tue Nov 20 14:35:04 GMT 2007
```

Figure 14-7: Postdata oddsline report, Folkestone 2:40, 20/11/2007

Indeed this is the case, and all three of the top ranked runners are overlays.

Following this race live through the commentary on the radio, we hear all the contenders backed by the oddsline filling the first three positions 6 fences from home, *Cossack Dancer, Inaro, FieldsOfClover,* although as the commentator says "They're all in a heap". This is perhaps not the ideal stage in a hotly contested national hunt handicap to be in the lead since making the running from the front is the tough way to win such races, and often best suited to animals that are a class ahead of the competition. *Cossack Dancer* continues to do well, but sounds like it is only a matter of time until it is caught by one of the chasing pack.

Soon enough, the second last of the overlays, *Inaro* drops away. Coming to the last fence, it's *Cossack Dancer* from *HereComesStanley. Cossack Dancer,* ridden by Mattie Bachelor puts in a great jump at the last, landed running. Returned the winner at an SP of 22/1, or 38.0 for us. An exciting finish and a great result.

However, if we are waiting for the oddsline strategy to hit the right race type – a hotly contested handicap at approximately 7/1 + against the field, something is surely wrong – it's great that if the oddsline is indeed suited to such contests, but then we should be eliminating the races that do not fit.

The rest of the day does not go so well, despite some opportunity through overlays at Wolverhampton, and a steady stream of bets flows out without much return – which is of course the downside of the situation described for Cossack Dancer's race, i.e. more volatility.

So, overall on the day, for both systems combined, a loss of -£26.60 is incurred, after a commission reduction of approximately 3.9% on all winnings. It was a bad day for the Equiform oddsline, reducing the profit shown by this one so far (despite finding 3 winners) and a good day for the Postdata oddsline, despite finding only one winner. But overall, we are poorer.

Table 14-3: P&L by Oddsline Strategy, November 20th 2007

Oddsline	Date	bets	wins	P&L	Cum.P&L
Equiform	20/11/2007	37	3	-83.35	87.81
Postdata	20/11/2007	25	1	65	-29.8

Wednesday 21st November

4 meetings today: Hexham, Lingfield, Warwick and Kempton (All Weather evening racing).

"We have had an unbelievable amount of rain since the weekend and it is going to be very testing," said the clerk of the course at Hexham.

Should make for some long priced contenders perhaps? The field sizes are up today, so that combined with a change in the weather is likely to make matters more volatile.

Let's see what happens, and cut straight to the chase, as it were, by curtailing the notes to look straight at our end of day position:

Table 14-4: P&L by Oddsline Strategy, November 21st 2007

Oddsline	Date	bets	wins	P&L	Cum.P&L
Equiform	21/11/2007	62	8	-13.95	73.86
Postdata	21/11/2007	52	3	-138.69	-168.49

It certainly was a volatile day - with combined automated losses of over £150 on both strategies. The main culprit was the Postdata oddsline accounting for over 90% of losses.

So what went wrong?

Firstly, let's consider a factor that may apply to both strategies (and is worth bringing up regarding timing for automation in general): properly accounting for recent changes in the going - where going is a variable that is relied upon for determining the oddsline. For example, if oddslines are automatically generated the day before racing, there rests the possibility that the factors for going are incorrect by the time that racing starts.

That aside, there are further observations that are pertinent. Profitability is in many ways all about price – for the rules applied to both oddsline strategies in particular, we are placing inherent emphasis on the accuracy of the oddsline versus the market. For an oddsline where no testing has been done on the accuracy of the derived prices versus the results, we are perhaps asking for trouble. Previously we observed that the method used for pricing for the Postdata oddsline ties it to relatively tight range of prices, being based more or less on integer scores, in the approximate range of 1 - 12. The Equiform oddsline however uses a wider ability range that is reflected in prices. Thus, even where both oddslines agree on the top rated horses within the race, the tissue prices can mean an overlay for one oddsline and "no bet" for the other. This is demonstrated by one of the earlier races of the day at Hexham, in the 12.40

```
Equiform:

Market timestamp:       12:35:03.725
12:40, Hexham, Chip N Pin, 5.03, 5.0
12:40, Hexham, Pagan Starprincess, 8.55, 9.2
12:40, Hexham, La Vecchia Scuola, 8.93, 18.0
12:40, Hexham, Kentucky Boy, 11.24, 5.9
12:40, Hexham, Bayonyx, 11.36, 8.6
12:40, Hexham, Slavonic Lake, 13.7, 14.5
12:40, Hexham, Act Sirius, 13.89, 11.0
12:40, Hexham, Rainbow Flame, 14.08, 32.0
12:40, Hexham, Acapulco Bay, 22.22, 70.0
12:40, Hexham, Impact Zone, 22.73, 90.0
12:40, Hexham, Find Me, 34.48, 14.0
12:40, Hexham, Danish Rebel, 41.67, 42.0
```

```
12:40, Hexham, Heavens Gates, 43.48, 55.0
12:40, Hexham, Zoren, 166.67, 24.0
12:40, Hexham, Waiheke Island, 250, 210.0
12:40, Hexham, Malguru, 251, 1000.0
ALL RUN

Postdata:

Market timestamp:       12:35:03.976
12.40, Hexham, Chip N Pin, 8.64, 5.0
12.40, Hexham, Pagan Starprincess, 10.56, 9.2
12.40, Hexham, Slavonic Lake, 11.88, 14.5
12.40, Hexham, Heaven's Gates, 11.88, 55.0
12.40, Hexham, Bayonyx, 11.88, 8.6
12.40, Hexham, La Vecchia Scuola, 13.57, 18.0
12.40, Hexham, Impact Zone, 13.57, 95.0
12.40, Hexham, Act Sirius, 13.57, 11.0
12.40, Hexham, Acapulco Bay, 15.83, 70.0
12.40, Hexham, Kentucky Boy, 15.83, 5.9
12.40, Hexham, Find Me, 15.83, 14.0
12.40, Hexham, Danish Rebel, 19.00, 42.0
12.40, Hexham, Rainbow Flame, 31.67, 32.0
12.40, Hexham, Waiheke Island, 47.50, 210.0
12.40, Hexham, Malguru, 95.00, 1000.0
12.40, Hexham, Zoren, 95.00, 24.0
```

Figure 14-8: Oddslines compared for Folkestone 12:40, 21/11/2007

Here the Equiform and Postdata oddslines both agree on the top 2 rated horses in the field. However, the method we have devised to create tissue prices for Postdata, based on a range of integer scores, means that the range of prices runs from 8.64 to 95.0, whereas the range of scores for the Equiform oddsline run from 5.03 to 251. That range is more reflective of the market range and is enough to establish that *Pagan Starprincess* is an overlay for Equiform at decimal odds of 9.2, but a no bet for Postdata, despite the predicted finishing order for both strategies being the same. Effectively the method we are using for Postdata is always saying that horses are more closely grouped in terms of ability than is perhaps the case. In certain race types, this will be more true than others, such as handicaps, but even in the case of handicaps it is not a universal truth.

Thus, as we have pointed out already, we are learning that the Postdata oddsline is not to be implicitly trusted without further testing and refinement, despite some interesting results.

Aside from the particularly poor performance of the Postdata oddsline, today has happened to coincide with the busiest day of the production run so far. With larger fields, together with more races, there are inevitably more overlays and volatility. For example, the 2.50 at Warwick with 18 runners carried 6 bets from each oddsline, and was therefore responsible for a loss of £60 alone. In fact, both strategies combined place almost double the number of bets than the previous day. Thus if either oddsline is performing particularly badly on the day – due to factors such as changes in going - this has a magnified effect on profit and loss on this day. Only time – or rather, sufficient testing over a sufficient number of trials - can tell if that is a reasonable assumption.

Thursday 22st November

The day starts off with the 12:40 at Market Rasen - changed to *Market_Rasen* in the world of automated betting, so we can match data where courses originate from different sources.

As for the race itself, it's another novice hurdle, as is often the convention for the opening race on a National Hunt card. This is the type of race we have seen so far this week that can throw up big priced winners, although at the other extreme of the rankings, it's usually the FAV or second FAV that have the best record of success.

To prove the case, *Balamory Dan* wins at 28/1, with a favourite, *Calgary Bay*, unwilling to pass him in the home straight. Unfortunately, *Balamory Dan* was not in the frame for either oddsline ranking.

For the rest of the day's racing, many bets are again selected by the strategies in the big fields. This characteristic, which went against us so badly yesterday, now comes good, particularly in the 2.50 at Hereford as *Harbour Breeze* lands the race at decimal odds of 16.5. Better still, *Harbour Breeze* is a qualifier for both oddslines, thereby netting a profit of £135.

This puts us back in front for the day at this point by £122, but the profit situation is eroded substantially by the end of afternoon racing. An assortment of mediocre races fielding contenders lacking in form fill the latter races of the day. Profit for the day is a meagre net 18.92 going into the Wolverhampton evening meeting, as shown below:

Betting P&L

colin magee 22-Nov-2007 16:30

Period: Just For Today

(relates to event settlement date)

2007-11-22 00: to 2007-11-22 16: Get P & L

(yyyy-mm-dd hh:mm) (yyyy-mm-dd hh:mm)

Download to Spreadsheet ?

Horse Racing: GBP18.92 Total P&L: GBP18.92

Horse Racing Showing **1 - 20** of 20 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Winc 22nd Nov : 2m Hcap Chs	22-Nov-07 16:00	22-Nov-07 16:10	-20.00
Horse Racing / Here 22nd Nov : 2m1f NHF	22-Nov-07 15:50	22-Nov-07 15:58	-20.00
Horse Racing / MrktR 22nd Nov : 2m1f Hcap Hrd	22-Nov-07 15:40	22-Nov-07 15:48	-10.00
Horse Racing / Winc 22nd Nov : 2m6f Nov Hrd	22-Nov-07 15:30	22-Nov-07 15:45	-20.00
Horse Racing / Here 22nd Nov : 3m1f Hcap Chs	22-Nov-07 15:20	22-Nov-07 15:30	-30.00
Horse Racing / MrktR 22nd Nov : 2m4f Hcap Chs	22-Nov-07 15:10	22-Nov-07 15:18	-20.00
Horse Racing / Winc 22nd Nov : 3m3f Hcap Chs	22-Nov-07 15:00	22-Nov-07 15:12	16.34

Horse Racing / Here 22nd Nov : 2m4f Hcap Hrd	22-Nov-07 14:50	22-Nov-07 14:56	126.51
Horse Racing / MrktR 22nd Nov : 2m6f Hcap Chs	22-Nov-07 14:40	22-Nov-07 14:52	19.32
Horse Racing / Winc 22nd Nov : 2m6f Hcap Hrd	22-Nov-07 14:30	22-Nov-07 14:38	-15.00
Horse Racing / Here 22nd Nov : 2m3f Hcap Chs	22-Nov-07 14:20	22-Nov-07 14:31	52.85
Horse Racing / MrktR 22nd Nov : 2m6f Beg Chs	22-Nov-07 14:10	22-Nov-07 14:19	13.45
Horse Racing / Winc 22nd Nov : 2m5f Nov Chs	22-Nov-07 14:00	22-Nov-07 14:06	-15.00
Horse Racing / Here 22nd Nov : 2m1f Sell Hrd	22-Nov-07 13:50	22-Nov-07 13:56	-20.00
Horse Racing / MrktR 22nd Nov : 2m6f Hcap Hrd	22-Nov-07 13:40	22-Nov-07 13:48	-30.00
Horse Racing / Winc 22nd Nov : 2m Hcap Hrd	22-Nov-07 13:30	22-Nov-07 13:37	50.45
Horse Racing / Here 22nd Nov : 2m Hcap Chs	22-Nov-07 13:20	22-Nov-07 13:27	-20.00
Horse Racing / MrktR 22nd Nov : 2m3f Claim Hrd	22-Nov-07 13:10	22-Nov-07 13:18	-10.00
Horse Racing / Here 22nd Nov : 2m1f Mdn Hrd	22-Nov-07 12:50	22-Nov-07 12:57	-10.00
Horse Racing / MrktR 22nd Nov : 2m3f Nov Hrd	22-Nov-07 12:40	22-Nov-07 12:48	-20.00

Profit and Loss is shown net of commission.

All times are UKT [?] unless otherwise stated.

Figure 14-9: Interim Account P&L for 22/11/2007

Nothing that has happened today has improved our overall position for the week, or done much to redeem our view on the unreliability of the Postdata oddsline, despite it showing a slight profit on the day.

Whether it is the performance of the oddslines, the poorer quality racing, the switch from flat to the jumps, the shorter nights, colder, wetter weather, or all of the above, following the progress of the strategies is at this point a miserable business. At least we can console ourselves with the fact that automation means none of this action *has* to be followed.

Off to Wolverhampton for the evening. Could that be the tonic to ease a depression?! Perhaps a better tonic would be Royal Ascot, the July meeting at Newmarket, Glorious Goodwood, or the York Ebor meeting? No, *Woolybags* it is, horses of the lowest rating running in circles on a Polytrack surface under floodlights.

For our automated strategy, all races come alike. After a poor start there is some good news in the 8:50 for Equiform ratings. *Carcinetto* which is a marginal overlay at 7.0 versus the tissue price of 6.99 comes home for us. That somewhat compensates for all the losers at Wolverhampton in the earlier races – but not quite.

It's worth commenting on the decision to bet on *Carcinetto* being very marginal. At a single bet level, many decisions can seem this way in an automated strategy - a lot of our wins have been by marginal distances, or we have narrowly missed out on good winners by the same narrow margin, or by applying the oddslines criteria of the top third, rounded down.

Again, the key factor is the number of races over which we are trialling, and that in the long run such near misses or marginal victories will be "smoothed out". This highlights

the difference with a manual betting strategy where near misses can affect the psychology and encourage the bettor to chase losses, or "lucky" wins which can encourage overconfidence and produce bets that would otherwise not have been made. The automated strategy keeps the ideal "level head", as it were. As we can see from keeping a diary, it is down to the punter to maintain an even keel if following it!

The aptly named *Following Flow* is the last bet of the day from the oddsline, a potentially big winner at 28.0 on the exchange, would put us nicely in profit, but it is not to be…

We end the day down in cash on both oddsline strategies, ironically with exactly the same number of bets placed, at 51 each.

Table 14-5: P&L by Oddsline Strategy, November 22nd 2007

Oddsline	date	Bets	wins	P&L	cum.P&L
Equiform	22/11/2007	51	5	-41.5	32.36
Postdata	22/11/2007	51	4	-18.26	-186.75

This reflects our worst position to date. The Postdata oddsline looks in dire need of modification before the losses start to consume more profit potential. We have already pointed out that the oddsline tends to do best in competitive handicaps, so that could be something to look into for the adjustment. At any rate, we cannot be comfortable with the current volume of bets, and must think either about reducing the stake or the number of races under consideration.

In the meantime, we will give at least a week's solid run to both strategies. There is also the hope for the remaining 2 days that - with the quality of racing generally picking up on a weekend - the form factors used in particular by the Postdata oddsline may be more significant. A quick glance at the fixtures shows us that tomorrow we have the start of an important Ascot meeting, as well as proper meetings for "the winter game" at Kelso and Exeter – and, of course, Wolverhampton in the evening. Not world class racing, but a sure sign that we are into the National Hunt season proper

Friday 23rd November

The day's racing again contains some bigger fields and inevitably an increased number of selections from the oddslines - but we draw a blank.

Wolverhampton in the evening does not improve matters, and the last three races of the meeting are abandoned. Our program report for the evening of course shows that there are non-runners in every one of those races – results may not be going our way but the implementation is working correctly.

Having observed many of the different ups and downs already with following this strategy through the week's diary, there is little more to note at this point other than the overall position at the end of the day:

Table 14-6: P&L by Oddsline Strategy, November 23rd 2007

Oddsline	Date	bets	wins	P&L	cum.P&L
Equiform	23/11/2007	42	5	-43	-10.64
Postdata	23/11/2007	37	2	-110	-296.75

The figures speak for themselves – at the same time, this is a very short time horizon. What is worth noting is that we need a resilient bank - even for staking £5 per bet - if operating this strategy.

Saturday 24th November

A much better quality day's racing - as is usually the case on a Saturday.

Good jumping cards at Haydock and Ascot, and Huntingdon. Even the all weather at Lingfield boasts a much better Saturday card, with better horses and prize money than in the midweek. Only Wolverhampton this evening is a less appealing spectacle.

In the few tests done prior to our production run, the homebrew "Racing Post Postdata" oddsline has done much better on a Saturday than weekdays, and can be a profitable method, even if backing overlays in all races. We saw evidence of this in the live test on Saturday October 13th.

Generally, better quality racing means higher prize money and horses that are more likely to turn up in reliable form, prepared for the race. Since the oddslines are based on historic performance, we would expect them to do better when the horses are being prepared to their best.

This pans out to be true enough through the day. Mostly the balance from both strategies is positive rather than negative, although it is still a case of a few steps forward and a few back.

A few poor results towards the end of the "proper" days' racing, i.e. before the evening meeting at Wolverhampton, see the profit margin reduced to £10.

The evening racing at Wolverhampton, ironically, rounds off the week nicely for us by producing a winner for the Equiform oddsline in *Moment of Clarity* in the last race, at a starting price of 20/1 and a Betfair price taken by the Equiform oddsline of 30.0.

Table 14-7: P&L by Oddsline Strategy, November 24th 2007

Oddsline	Date	bets	wins	P&L	cum.P&L
Equiform	24/11/2007	57	7	117.75	107.11
Postdata	24/11/2007	44	5	26.5	-270.25

In terms of overall results, we finally have a profit to report on both oddslines. Postdata again does better on a better quality day's racing, as suspected, with both oddslines

appreciating the Saturday racing. Equiform regains the ground it had lost, and goes into overall profit to end the week.

Conclusions and Adjustments at end of week 1

We have stated on a number of occasions over the first week that the Postdata oddsline is untested, which causes a couple of major problems. Firstly, the tissue prices are based on a hypothesis rather than backtesting against past results, and secondly we have no idea in which races the Postdata oddsline may be strong or weak.

All of which goes to show, as we could have predicted, that it is better to test a strategy against a results database *before* implementing it (a process we discuss in the next Chapter).

With regard to the Equiform oddsline, which has been thoroughly backtested against historic results, we are at least looking at a profit, albeit insignificant compared to the number of bets placed over the week - but enough to encourage us to persevere for the final week of the production run.

Of course, the fact that the Equiform oddsline is tested does not mean it will always produce profits – there are good and bad weeks, months and even seasons. In spite of backtesting, there is always the risk of a significant shift in market (and therefore pricing) conditions ahead of the tested oddsline. This can mean that the same strike rate (i.e. winners to runners ratio) is maintained but if the average price decreases over time, a profitable strategy becomes unprofitable. The reliance on the fact that the oddsline is tested simply means that the strategy has already been optimised against past results, and there is little advantage to be gained by tinkering with it over any given 2 week period.

In the meantime, we are faced with trying to optimise what we have already as far as the Postdata oddsline is concerned. Although the overall results are not impressive, there are elements of the strategy that made it attractive in the first place. We have seen during the week that the indiscriminate application of the Postdata oddsline to every race is a scattergun approach that sometimes works and sometimes doesn't. As we consider how to adjust the strategy, let's focus on what has worked instead.

Where there is a lot of appropriate form to go on, the Postdata oddsline has more factors that can be included in any score, for each contender in a race. Wherever it is taking into account more factors about a contender's proven ability to perform on a given day, it stands to reason that the oddsline should be more predictive. Where there are more unknown factors, there is more margin for error in assessing any contender's chance. We also have to consider how predictive and how profitable (two different questions entirely, relating to strike rate and price obtained) the use of the type of form factors considered in the Postdata oddsline will be.

We have already hypothesised that Postdata may be doing less well on the All Weather at this time of year due to form factors being less powerful with the poor quality or racing, and possible factors related to the start of the All Weather season.

So, first, let's see if results vary according to the code of racing, breaking down all bets that we have captured automatically during the course of the production run.

We can therefore produce a MySQL query using the database tables we have described so far which goes through all our bets by **marketId**, joining with our **`races`** database table, where the Betfair race description is kept. We group all races with the description of hurdles, chases and National Hunt Flat into one category, and all others (being all weather races at this time of year) into another.

The query then groups the results for each day in terms of number of bets, number of winners, and daily profit and loss. We show the split of all results by racing code, as in *Table 14-8*, and *Table 14-9*. In Table *14-10*, we look at summary statistics and the return on investment for both sets of data:

Table 14-8: P&L Postdata All Weather Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
19/11/2007	24	1	-60	-60
20/11/2007	11	0	-55	-115
21/11/2007	21	2	2.5	-112.5
22/11/2007	13	0	-65	-177.5
23/11/2007	4	1	28	-149.5
24/11/2007	20	1	-52	**-201.5**

`Total P&L = - £201.50`

Table 14-9: P&L Postdata National Hunt Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
2007-11-18	33	3	-18.3	-18.3
2007-11-19	9	1	-16.5	-34.8
2007-11-20	14	1	120	85.2
2007-11-21	31	1	-141.19	-55.99
2007-11-22	38	4	46.74	-9.25
2007-11-23	33	1	-138	-147.25
2007-11-24	24	4	78.5	**-68.75**

`Total P&L = - £68.75`

Table 14-10: Postdata P&L breakdown by race code, week 1

Racing code	#Bets	Outlay £	Return	ROI
AW	93	465	-201.5	-43.33%
NH	172	860	-68.75	-7.99%

A negative 43% on turnover as opposed to a negative 8% on turnover is evidence that our suspicion regarding All Weather racing has some basis.

Despite the fact this test has only been over one week, there are practical concerns about the strategy that are also difficult to ignore.

For example, for all weather racing at this stage of the season, we know we are considering races in which horses of the poorest form are competing. In addition, since it is the start of the All Weather season, the horses are often unproven on prevailing conditions, be it course, distance or going, and they lack relevant recent form. Given that the Postdata oddsline is based on the value we attach to the Postdata ticks and crosses as useful measures of form (and specifically form according to conditions of the race in question), we can see that there may be problems with the value of all weather assessment at this time of year. Perhaps there are other measures that could be used to improve the analysis, or a different weighting that could be given to the measures (or variables) already present – but in its current form, it is too risky.

The other common factor we have discussed in the diary, using the same reasoning as above, is that handicap races generally include horses with more relevant historic form than non-handicaps (in the non-handicap category, we have novices and maidens, for example). This means that the Postdata factors have more appropriate form to assess, and can match suitability to race conditions with greater frequency.

We also know that the Postdata oddsline has done well in competitive racing where the oddsline tends to conform to the "shape" of the market itself (i.e. there is a tighter range of prices) and we are likely to get overlays on our top rated selections, as well as those further down the list. In races with a wider range, the method of scoring the Postdata oddsline means we have missed out on top ranked selections being overpriced compared to their market chance. Whilst this is a potential flaw in the way we are pricing, for the time being we are looking at how to optimise returns on the tissue we are already working with.

So let's see what happens if we further break out the results of handicaps across all racing codes, since handicaps generally (although not always) fit the bill in terms of competitive racing where there is lots of known form.

We can therefore produce a MySQL query similar to that used for splitting out racing codes, but this time concentrating on the race type, again using the Betfair race description from the **`races`** database table, using the following syntax in our query:

```
INNER JOIN races ON marketId = Betfair_id WHERE (description LIKE "%Hcap%")
```

The query then groups the results for each day in terms of number of bets, number of winners, and daily profit and loss. We show reformatted query results for the week in *Table 14-11*, for all handicap races bet by the Postdata oddsline strategy over the first week of the production run.

Table 14-11: P&L Postdata Handicap Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
18/11/2007	21	2	33.5	33.5
19/11/2007	19	2	-6.5	27
20/11/2007	11	1	135	162
21/11/2007	27	1	-47.5	114.5
22/11/2007	35	4	61.74	176.24
23/11/2007	23	1	-67	109.24
24/11/2007	26	2	-12.5	**96.74**

Total P&L = £96.50

So there is some vindication of our observations to date, and we can again see the potential benefit of early monitoring. Our otherwise poor results – if considering every horse in every type of race - suddenly look more respectable. We are in profit to the tune of £96.50 over the week for fewer bets – almost 50% fewer – thus improving the overall return on investment. Our casual hypothesis about the Postdata oddsline performing better in handicaps has empirical proof.

So, what to do with this information? Given the earlier findings on the profitability of National Hunt racing over All Weather, and that generally the evening (i.e. all weather) meetings decimated profits from the Postdata oddsline, let's also split the P&L from handicap races into different codes: All Weather racing and National Hunt.

Table 14-12 shows us the performance for all weather racing only.

Table 14-12: P&L Postdata Handicap AW Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
19/11/2007	14	1	-10	-10
20/11/2007	4	0	-20	-30
21/11/2007	12	1	27.5	-2.5
22/11/2007	9	0	-45	-47.5
23/11/2007	3	1	33	-14.5
24/11/2007	10	0	-50	**-64.5**

Total P&L = - £64.50

The performance in all weather handicaps was actually negative, and therefore reduced our performance in handicaps overall. If we look at the total number of bets placed by the Postdata strategy on all races over the first week, it is 275, producing a loss of £270. The performance on all weather racing alone is 52 bets producing a loss of £64.50. So there is nothing in there to indicate that by keeping this stream of bets going we would be any better off.

Let's look next at National Hunt, and break down into the two types of handicap in national hunt, handicap hurdles and handicap chases, *Table 14-13* shows Handicap Hurdles only, *Table 14-14* Handicap Chases only.

Table 14-13: P&L Postdata Handicap Hurdle Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
18/11/2007	14	1	-24	-24
19/11/2007	5	1	3.5	-20.5
20/11/2007	2	0	-10	-30.5
21/11/2007	7	0	-35	-65.5
22/11/2007	12	2	101.64	36.14
23/11/2007	11	0	-55	-18.86
24/11/2007	6	0	-30	**-48.86**

Total P&L = - £48.86

Surprisingly, the performance over all handicap hurdles was also negative. However, it is clear that the operation of the strategy identified some good individual winners. The performance on handicap hurdles alone is 57 bets producing a loss of £48.86. If 57 bets is a total outlay of £285, this is a negative return on investment of over 15%. In pure P&L terms we might choose to exclude this category for the next week; on the other hand, since the data is relatively small, the belief in the basis of the hypothesis itself is still an important consideration in the decision on whether to take this category forward.

Let's look next at Chases:

Table 14-14: P&L Postdata Handicap Chase Bets only, week 1

Date	#Bets	#Wins	Return £	Cum(P&L)
18/11/2007	7	1	57.5	57.5
20/11/2007	5	1	165	222.5
21/11/2007	8	0	-40	182.5
22/11/2007	14	2	5.1	187.6
23/11/2007	9	0	-45	142.6
24/11/2007	10	2	67.5	**210.1**

Total P&L = £210.10

Having excluded other handicap types to find losses, it is no surprise that this is the most profitable category, and by some margin.

A total of 53 bets were placed, or an outlay of £215, for a return of £425.10. That's a 97% positive return on investment. Of course, the data are relatively few, and our results are subject to the influence of individual winners such as *Cossack Dancer*, which won at a price of 38.0 on the 20th November. However, even if we take this result out,

the strategy still shows a profit overall. Furthermore, the strategy shows other good indicators, with a daily profit shown on 4 of the 7 days, and a reasonable strike rate for long priced winners (at 11%, versus the all weather handicap strike rate of 5.7% and the hurdles handicap strike rate of 7%).

Conclusions on Analysis of Postdata Oddsline Strategy Results:

We have seen that continuing with the untested Postdata oddsline on all races produces a large number of bets and is unprofitable overall. We surmise that there is little downside to modifying the strategy (not the oddsline itself, which may be also be useful but is a bigger task) in favour of a strategy which does not bet indiscriminately on every race.

We have therefore modelled the effects of changing the strategy by discriminating with regard to racing codes and race type - we have seen that all weather racing is to be avoided, and further, that handicap races (in particular, handicap chases) produced profitable returns.

Had we run such a model over the first week the oddsline would have been profitable. The plan now is to run the new model on as yet unknown events, over the week to come. In pure data terms, we had only one week's worth of data to work with, so it is difficult to form any hard and fast conclusions, indeed it could be argued this is a case of "overfitting".

On the other hand, it is useful to ask if the logic behind the model makes sense – and since the model was produced on the basis of reasoning first and foremost, rather than the data alone, we can answer in the affirmative. Also, the reasoning we have used is very particular to this time of the season, and we therefore expect it to be a transitory effect. Again, without further testing we cannot say for how long we might expect it to last or even if it is repeatable.

It may well be that considering historic form factors at this early stage of the National Hunt season, before new form lines have had a chance to establish themselves, is a phenomenon that is particularly profitable for a short time. Perhaps the factors used by the oddsline are underestimated by the market at this time of year? Again, on the basis of the first week's results there is little downside to limiting the number of events considered by the Postdata oddsline. The upside is that if we decide to run the revised strategy on the basis of the more profitable model that was hidden away within the results, it may again be profitable.

Whatever the case on the data and our reasoning, ultimately making any decision to change tack or modify a strategy is a call for the individual bettor to make – and we will take the risk to go with it.

Adjustments to Postdata Oddsline strategy:

Combining the two main conclusions from our analysis, we will now consider National Hunt handicaps only (excluding all weather races and any other National Hunt race types) for the Postdata oddsline for the second week of the production run.

Having left the automated strategy untouched for the first week, we make our only manual adjustment to the software configuration on the Saturday, after racing has finished.

There turns out to be an elegant solution to adjusting the whole strategy as we have defined it, with only a one line code change to one program.

There is no need to make an alteration to our betting program, or to the program that automatically schedules the betting program to execute 5 minutes before the race based upon the events schedule (represented by the file `events_schedule` in the examples). Neither do we need amend the `get_betfair_races.pl` program which retrieves all events – which is just as well, since the Equiform strategy will continue to bet all races. The only program we have to amend is the one shown in *Example 7-1*, which creates the events schedule in the first place, since the change in strategy only modifies the events under consideration. Once we have a new events schedule, all the other programs can work "as is".

Recall that this program includes within the Perl code a MySQL query to our **`races`** table – to retrieve all races in the UK with the current date, ordered by the time of the race. Instead, we simply need to modify the MySQL `SELECT` statement in order to choose those races with the `WHERE` clause applied to the Betfair race description to select "Hcap Hrd" or "Hcap Chs" only. Everything else works as before, including the cron jobs which do not need to be modified either. The only difference is that the events schedule that is written to the directory running the Postdata oddsline will contain fewer events for every day's racing.

Subsequently, when the dynamic scheduling program picks up the new events schedule, it will create scheduled tasks for Handicap Hurdles and Handicap Chases only, still running 5 minutes before each race.

Week 2:

25th November – 2nd December 2007

Unfortunately we have a negative overall P&L position to start the week. Having achieved our objective of understanding exactly how the oddslines are operating during the first week, we are now mostly concerned that they turn a profit. As such, we no longer have a straight comparison of all races in the Equiform oddsline to the Postdata oddsline, although both strategies are still operating in parallel and share some races in common. The well tested Equiform oddsline is in profit for the first week, and we are leaving it to run for the second week, untouched. The untested Postdata strategy has now been forced to concentrate on National Hunt handicaps only, where we found it to be generally profitable, after racking up a 20% loss on turnover on all bets in the first week.

Sunday 25th November

A day of jumps fixtures at Towcester, Plumpton, and the feature card at Aintree.

Original Fly, top rated in the last race at Aintree, wins at decimal odds of 25.0 and caps a great day for the Equiform oddsline, as shown in *Table 14-15*.

Table 14-15: P&L by Oddsline Strategy, November 25th 2007

Oddsline	Date	bets	Wins	P&L	cum.P&L
Equiform	25/11/2007	49	9	342.5	449.61
Postdata	25/11/2007	20	0	-100	-370.25

A number of overlays at good prices see our most profitable day yet. This is diminished by the performance of the Postdata oddsline strategy, which, despite the adjustments of the previous week, yields no winners but still racks up 20 bets. Ironically, there are a couple of "missed winners" from the race types that are now excluded from the adjusted strategy, which would have led to a more profitable situation. Still, we have made one change already and tested it, so we are going to stick with it. The performance of the Equiform oddsline gives us some small cushion of profit, so after the first day on the second week, we have retrieved the situation somewhat.

Monday 26th November

After more all weather action at Lingfield in the 12.30, the day starts in earnest for both oddsline strategies with the 12.40 at Ayr, which is a 2 mile 5 furlong handicap hurdle, and therefore a qualifier for our modified Postdata oddsline strategy.

```
################################  NEW ODDSLINE  ################################

Ayr, 12.40
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       12:35:03.005
12.40, Ayr, Double Ells, 8.91, 5.7
12.40, Ayr, Mrs O'Malley, 10.89, 11.5
12.40, Ayr, Tobesure, 10.89, 18.0
12.40, Ayr, Named At Dinner, 12.25, 11.0
12.40, Ayr, Millie The Filly, 12.25, 28.0
12.40, Ayr, Greenock, 14.00, 18.0
12.40, Ayr, Topaz Lady, 14.00, 10.5
12.40, Ayr, Native Coll, 16.33, 23.0
12.40, Ayr, Windygate, 16.33, 12.0
12.40, Ayr, Teviot Brig, 16.33, 14.5
12.40, Ayr, Stravaigin, 16.33, 21.0
12.40, Ayr, Persian Prince, 16.33, 9.4
12.40, Ayr, Cupid's Mission, 24.50, 55.0
12.40, Ayr, Bene Lad, 24.50, 16.5
12.40, Ayr, Presenting Edward, 98.00, 200.0
ALL RUN

Bet top third overlays:
Selection:   12.40, Ayr, Mrs OMalley, 20708178, 2313604
Bet placed: 4053381611, B, £5.00,  11.5
Selection:   12.40, Ayr, Tobesure, 20708178, 57480
Bet placed: 4053381668, B, £5.00,  18.0
```

```
Selection:   12.40, Ayr, Millie The Filly, 20708178, 2206599
Bet placed: 4053381749, B, £5.00,  28.0

Oddsline strategy finished at systime: Mon Nov 26 12:35:04 GMT 2007
```

Figure 14-10: Postdata Oddsline for 12.40 Ayr, 26/11/2007

Milly The Filly, wins the race at decimal odds of 28.0, ranked fifth in the Postdata oddsline within a field of 15 that go to post.

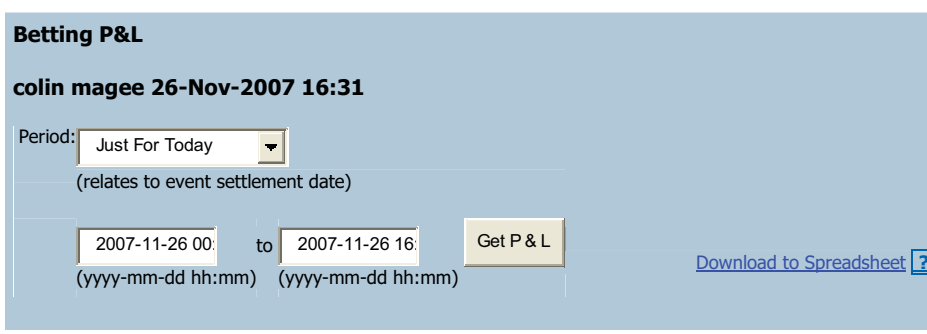
At last, this race provides some vindication for the strategy to go with National Hunt handicaps only with the Postdata system.

On the whole, it turns out to be a great day for both programs, as we swing into profit all round. Lots of good priced winners and the oddslines find most of them. The handicap strategy pays particularly well. As with the previous day, it so happens that a strategy including all National Hunt races, not just handicaps, would have been just as, if not slightly more profitable for the Postdata oddsline, due in part to finding the winner of "The Bumper" at decimal odds of 15.0.

In the meantime, outside the world of the hypothetical, for the strategy we have chosen, the upside is that there is a much greater return on investment for the few races we have played, with 6 races considered as handicaps, and the winner found in 3 of them. Overall, it is better to produce the same profit from betting in 6 races, or even slightly less, than in 22 races since our risk is significantly reduced, and the return on investment increased.

The Equiform oddsline finds the majority of decent priced winners that were missed by our revised Postdata line, in all categories of racing. Over the long run, it is all about better pricing for horses' true chances, based on significant backtesting – so we consistently find winning overlays. Today just happens to be a day for longer priced winners, as was yesterday. We cannot expect every day to be like this, but these are the critical days for the long term profitability of the strategy.

So, with both oddslines firing in winners, at double digit prices, we have our best day yet. *Figure 14-11* shows the account balance at the end of the day



Horse Racing: GBP444.52 Total P&L: GBP444.52

Horse Racing Showing **1 - 20** of 21 markets

Market	Start time	Settled date	Profit/loss (GBP)
Horse Racing / Ludl 26th Nov : 2m NHF	26-Nov-07 15:50	26-Nov-07 16:09	52.80
Horse Racing / Ayr 26th Nov : 2m NHF	26-Nov-07 15:40	26-Nov-07 15:55	-10.00
Horse Racing / Ling 26th Nov : 1m2f Hcap	26-Nov-07 15:30	26-Nov-07 15:42	-5.00
Horse Racing / Ludl 26th Nov : 3m Hcap Hrd	26-Nov-07 15:20	26-Nov-07 15:33	168.00
Horse Racing / Ayr 26th Nov : 2m4f Hcap Chs	26-Nov-07 15:10	26-Nov-07 15:21	41.28
Horse Racing / Ling 26th Nov : 7f Hcap	26-Nov-07 15:00	26-Nov-07 15:14	-5.00
Horse Racing / Ludl 26th Nov : 2m4f Beg Chs	26-Nov-07 14:50	26-Nov-07 15:01	40.32
Horse Racing / Ayr 26th Nov : 2m4f Nov Hrd	26-Nov-07 14:40	26-Nov-07 14:48	62.40
Horse Racing / Ling 26th Nov : 1m Hcap	26-Nov-07 14:30	26-Nov-07 14:47	-15.00
Horse Racing / Ludl 26th Nov : 2m4f Hcap Chs	26-Nov-07 14:20	26-Nov-07 14:31	8.64
Horse Racing / Ling 26th Nov : 1m Mdn Stks	26-Nov-07 14:00	26-Nov-07 14:20	5.28
Horse Racing / Ayr 26th Nov : 2m Hcap Hrd	26-Nov-07 14:10	26-Nov-07 14:20	-35.00
Horse Racing / Ludl 26th Nov : 2m Sell Hrd	26-Nov-07 13:50	26-Nov-07 14:00	-15.00
Horse Racing / Ayr 26th Nov : 3m1f Beg Chs	26-Nov-07 13:40	26-Nov-07 13:51	115.20
Horse Racing / Ling 26th Nov : 7f Claim Stks	26-Nov-07 13:30	26-Nov-07 13:40	-5.00
Horse Racing / Ludl 26th Nov : 3m Hcap Chs	26-Nov-07 13:20	26-Nov-07 13:31	-20.00
Horse Racing / Ayr 26th Nov : 2m4f Nov Hrd	26-Nov-07 13:10	26-Nov-07 13:18	-10.00
Horse Racing / Ling 26th Nov : 6f Hcap	26-Nov-07 13:00	26-Nov-07 13:07	-15.00
Horse Racing / Ludl 26th Nov : 2m5f Nov Hrd	26-Nov-07 12:50	26-Nov-07 13:03	-10.00
Horse Racing / Ayr 26th Nov : 2m5f Hcap Hrd	26-Nov-07 12:40	26-Nov-07 12:49	105.60

Pages: 1 2 of 2 Pages

Jump to page:

Figure 14-11: Betfair Account P&L for 26/11/2007

A profit of £444.52 is the actual cash won on the day, with this being the balance with commission applicable to this account deducted. In terms of raw P&L for the system, the position is £265.50 in profit for the Equiform oddsline and £204.10 for the Postdata oddsline. It has taken a while, but it is days like this that more than make up for the losing ones.

Tuesday 27th November

Racing today at Kempton, Sedgefield and Wolverhampton.

We are soon back to the kind of moderate midweek fayre - being a combination of jumps meetings including races with very few runners and all weather poor quality fields - that characterised our first week. This produces similarly frustrating results through the day.

However, the damage that occurred during the first week is mitigated by the altered rules for the Postdata strategy, since the Postdata oddsline was the worst offender, so that it

avoids all races but handicaps. Unfortunately, the few handicaps that are considered also produce several bets but no winners. Such is to be expected with a strategy over a small window of observation – our speculation is that this will come right in the long run, as it did yesterday, and cover losing runs to leave us with a profit.

When the automated daily profit and loss data for each system has been retrieved, as shown in *Table 14-16*, we see that the damage has at least been limited on the Postdata oddsline, turning in a loss of £56 for the day, whilst Equiform manages a small profit (without any commissions applied), thanks to a winner in the last race at Wolverhampton.

Table 14-16: P&L by Oddsline Strategy, November 27th 2007

Oddsline	date	bets	wins	P&L	cum.P&L
Equiform	27/11/2007	40	5	8.04	723.15
Postdata	27/11/2007	15	1	-56.5	-222.74

Wednesday 28th November

A ragbag of talents on display at the three National Hunt meetings up and down the country as the day starts off with various novice, claiming and selling events at Wetherby, Chepstow and Lingfield. Kempton All Weather comes along later in the evening. We can take some small comfort in the fact that the unmodified Postdata oddsline was not betting in any of these races.

Otherwise the results are variable all around.

In the 2.55 at Chepstow we have a classic case of both oddslines agreeing on top rankings in a large handicap, but with a short priced favourite, *Blue Splash*. *Blue Splash* wins, so in hindsight it is value at any price, but wins well by 3.5 lengths.

However, our strategy when short priced winners oblige rarely helps the profit and loss situation. Despite the short price, there is a more accurate estimation of probability based on its strong chances according to the Equiform oddsline and it is a bet (at a tissue of 3.23 versus a market price of 3.5). However, the return does not pay for the other 2 bets selected by our dutching strategy in the race, so all we are doing is limiting losses in this particular case.

The Postdata oddsline, on the other hand, does not select the winner as an overlay at all. As we observed after the first week, this oddsline generally has a much tighter range of prices. In fact, the method produces 10/1 the field, so the top rated selection is ignored completely. The knock on effect is that the other contenders in the market are all longer prices, due to the short priced favourite, and therefore backed.

Silver Ingott, which has been off the course for the best part of 3 years, wins the penultimate race of the day and rounds off a pretty poor day after Tuesday's highs.

Down £185 in total with only the last race of the day proper, a bumper, to come. *Rock Alliance* is an overlay at 42.0, and is the only bet that can save the day's profits.

Kempton produces no better, and it is the Equiform oddsline's turn to suffer, having played in most races during the day, to produce a whopping 63 bets from the 4 meetings, and a loss before commissions of £224.75 on the day. The Postdata oddsline produces similarly poor results, but having been limited to handicaps only, the damage, also, is limited. An overall loss of £49 before commissions, and 16 runners to 1 winner. Pro rata for the Postdata oddsline, in terms of strike rate and P&L, that is more or less the same performance as the Equiform oddsline. At least we can be certain our revised strategy has saved some losses today.

Thursday 29th November

At last, the ragbag of novice and maiden hurdle events –which is an important part of bringing NH horses into the game but gives little form for the bettor to go on – today play into the favour of the Equiform oddsline, as a long priced outsider, *Moonhawk* at 28.0 (not much bigger than the SP of 25/1) wins the first race at Carlisle, a Maiden Hurdle.

That profit puts the Equiform oddsline strategy in good shape and stays with us through the first few races, but a big field handicap chase with both oddslines firing in bets is off in the 3.10, producing a total of 7 bets with 2 contenders shared between both oddslines, and is a pivotal race for the day's profitability.

Listening to the closing stages, it is the favourite, ridden by AP McCoy, who jumps the last together with one of the contenders common to both oddslines, *Lazy But Lively*, neck and neck they go to the line. I don't hold much hope – supporting a horse against AP in a finish – and against the favourite. It's close, but *Lazy But Lively* wins it for us.

This has now the makings of a profitable day, turning around our fortunes from yesterday, and putting a bit back into the pocket of the revised Postdata oddsline. But some big fields left to go in the few races remaining, and a number of them handicaps, could yet dent our profitability, or extend it.

Oscar The Boxer, a common overlay for both oddslines, is eventually denied by less than a length in the 3.30 at Carlisle.

Serious damage indeed in a couple of subsequent races sees the Postdata oddsline lose £60, £25 and £30. The overall position is saved somewhat by Equiform which retrieves £25 or so in the next, in which the Postdata oddsline, now belying its earlier excesses, abstains on the race altogether.

The Postdata oddsline produced £22 profit, having been eroded somewhat from our mid-day position, and the Equiform oddsline came back strongly to finish at £162 up. We therefore end the day well ahead - almost, but not quite, recovering all yesterday's losses, with 2 days to go until the end of the production run.

Friday 30th November

The start of the Hennessy meeting at Newbury. Good quality racing is much more satisfying to bet on, even when it is automated. Following the racing is far more interesting. Meanwhile, the computer is doing all the work.

The Hairy Lemon is a common overlay and winner in the 1.30 at Newbury, which keeps both oddslines' heads above water for the day

A winner at Lingfield for Equiform in the 2.15 at decimal odds of 17.0 puts us ahead on the day, the Postdata oddsline limping along somewhat. The big handicaps are again very volatile for profit and loss, with a lot of overlays going in.

At the end of daylight racing, both oddslines are in profit, Postdata is finished for the day, with no more events in the schedule given the evening meeting is all weather, in profit to a small amount at £15.

Wolverhampton is left to come, which, if we calculate an average of 2 – 3 bets per race could yield around 15 more bets, and therefore has the potential to wipe out the overall profit on the day. Still, there would not be much damage even if all go down, and we may even extend the position in good shape for the final day of the trial.

In the evening at Wolverhampton, only Equiform is betting. There is a good start to the evening with winners in the first two races at decimal odds of 9.0 and 10.0 respectively, thereby shoring up profits for the day and indeed extends it. Equiform finishes the day at £114.

We are nicely set up for the final day of the production run, Hennessy Day at Newbury.

Saturday 1st December

There are 5 meetings in total for the final day of the production run, with quality racing all round, including the valuable Hennessy Gold Cup Handicap Chase at Newbury and the "Fighting Fifth" Hurdle at Newcastle.

There is the potential for a lot of bets, here, although we are playing the day's racing with a diminished Postdata oddsline strategy. A quick check in the morning of the events schedule for each strategy reveals that there are 11 qualifying handicap races, whereas the full schedule of 35 races will be considered by the Equiformratings strategy. Effectively 11 races will be considered by two strategies and 24 by one strategy alone.

An early start with the 12:10 at Newcastle. The first two races are both, in fact, handicaps, and we draw a blank, both with Postdata and Equiform. As we have seen, however, running a successful automated betting strategy is all about making profit than loss over the long run, and only 20 minutes later in the 1.0 pm at Newbury, we have a fine example of how the automated strategies can work in tandem, identifying "value":

JunctionTwentyFour is the Equiform overlay at market odds of 8.2.
Sir Bathwick is the Postdata overlay at market odds of 17.0.

The ground is extremely testing at Newbury, both horses slug out the finish, well ahead of the rest of the field, it is nip and tuck to the line. *Sir Bathwick* wins.

This is good news as the most profitable winner of the race for the account. Moreover, it means that the revised implementation of the Postdata strategy (which looked so risky in

its raw form during the first week) looks likely to be vindicated for the second week. Although somewhat dependent on the performance of the other races today, even at this stage we can conclude it has reduced risk and given us a fighting chance of profiting on the most promising race types.

By contrast, it has been a sluggish start to the day for the Equiform strategy, which has top or second rated a number of winners but not found sufficient value in the market to bet. At least, since the fields are small, the strategy is not risking too many bets on any particular race up to now, and winners from *Helen's Vision* at Newbury, and *KeepTheDreamAlive* in the 1.25 at Towcester reduce the losses, about 6 points down after a total of 16 bets, but keep the oddsline respectable with 25 races to go for the day – operating the strategy itself is a staying race not a sprint.

In the 2.05 at Newcastle, the Fighting Fifth, *Inglis Drever* wins a fantastic race to take the race for the third year in a row.

In the big race at Newbury, *Denman*, under a huge weight in testing conditions, proves every bit as good as the hype surrounding him and lands the Hennessy in great style.

Denman is third in the Postdata oddsline ranking and missed out as an overlay, but he is top rated and deemed value by Equiform, and a bet goes in at 7.20 that saves overall profits on the race for the account as a whole, as a total of 7 bets are placed.

The 3.15 at Newbury is also a handicap, where *Kelrev* is top rated on the Postdata oddsline, deemed an overlay against tissue odds of 11.86, and duly wins. An SP of 14/1 is returned, with 22.0 taken on Betfair 5 minutes before the off, propelling the profit on the Postdata oddsline further ahead. The revised Postdata strategy is firmly in profit for the week.

On to the last handicap national hunt race of the day, to round off a great day's racing at Newbury in particular, with oddslines from the Equiform strategy and Postdata shown below in Figures *14-12* and *14-13* respectively.

```
Newbury, 3:45
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       15:40:04.063
3:45, Newbury, Poquelin, 7.58, 4.7
3:45, Newbury, Peacock, 9.09, 8.4
3:45, Newbury, Kings Revenge, 10.64, 15.5
3:45, Newbury, Cape Greko, 12.5, 32.0
3:45, Newbury, French Saulaie, 13.51, 25.0
3:45, Newbury, Kickahead, 13.7, NON_RUNNER
3:45, Newbury, Kanad, 20, NON_RUNNER
3:45, Newbury, Im So Lucky, 20.41, 32.0
3:45, Newbury, Tagula Blue, 20.41, 11.5
3:45, Newbury, Mon Michel, 21.74, 23.0
3:45, Newbury, French Opera, 25, NON_RUNNER
3:45, Newbury, Pevensey, 25.64, 9.0
3:45, Newbury, European Dream, 27.78, 8.6
3:45, Newbury, Ingratitude, 29.41, NON_RUNNER
3:45, Newbury, Mount Sandel, 31.25, 50.0
3:45, Newbury, Manhattan Boy, 32.26, 22.0
3:45, Newbury, Glinton, 55.56, 75.0
3:45, Newbury, Signs Of Love, 76.92, 9.6
NON_RUNNERS
```

```
Bet top third overlays:
Selection:   3:45, Newbury, Peacock, 20717542, 2221057
Bet placed: 4082717166, B, £5.00,  8.4
Selection:   3:45, Newbury, Kings Revenge, 20717542, 1117405
Bet placed: 4082717197, B, £5.00,  15.5
Selection:   3:45, Newbury, Cape Greko, 20717542, 853024
Bet placed: 4082717243, B, £5.00,  32.0

Oddsline strategy finished at systime: Sat Dec  1 15:40:04 GMT 2007
```

Figure 14-12: Equiformratings Program Report for 3.45 Newbury, 01/12/2007

```
Newbury, 3.45
API calls:
Login, GetMarkets, GetMarketPricesCompressed successful

Market timestamp:       15:40:03.283
3.45, Newbury, Poquelin, 12.67, 4.7
3.45, Newbury, King's Revenge, 12.67, 15.5
3.45, Newbury, European Dream, 14.25, 8.6
3.45, Newbury, Mon Michel, 14.25, 23.0
3.45, Newbury, Peacock, 14.25, 8.4
3.45, Newbury, French Opera, 16.29, NON_RUNNER
3.45, Newbury, Tagula Blue, 16.29, 11.5
3.45, Newbury, Kanad, 16.29, NON_RUNNER
3.45, Newbury, Kickahead, 16.29, NON_RUNNER
3.45, Newbury, French Saulaie, 19.00, 25.0
3.45, Newbury, Cape Greko, 19.00, 30.0
3.45, Newbury, Glinton, 19.00, 70.0
3.45, Newbury, Manhattan Boy, 22.80, 20.0
3.45, Newbury, Pevensey, 22.80, 9.0
3.45, Newbury, Ingratitude, 28.50, NON_RUNNER
3.45, Newbury, Mount Sandel, 28.50, 50.0
3.45, Newbury, I'm So Lucky, 28.50, 32.0
3.45, Newbury, Signs Of Love, 28.50, 9.4
NON_RUNNERS

Bet top third overlays:
Selection:   3.45, Newbury, Kings Revenge, 20717542, 1117405
Bet placed: 4082717028, B, £5.00,  15.5
Selection:   3.45, Newbury, Mon Michel, 20717542, 1233117
Bet placed: 4082717080, B, £5.00,  23.0

Oddsline strategy finished at systime: Sat Dec  1 15:40:04 GMT 2007
```

Figure 14-13: Postdata Oddsline Program Report for 3.45 Newbury, 01/12/2007

The common overlay to both oddslines is *King's Revenge*, so we are effectively having £10 on that at decimal odds of 15.5.

We are rooting for *Kings Revenge* all the way, but *Mont Michel*, our alternative, longer priced overlay, almost forgotten during the race, comes through in the final stages to

land the spoils at a price of 23.0 for the Postdata strategy. A good end to Postdata's efforts for the week, with a winning bet to go out on.

Meanwhile, Equiformratings has a bit of betting left to do before the end of the day and a meeting at Wolverhampton to get through. It turns out that this eats somewhat into the profit position from the end of day proper, although the same course was kind to us the day before, so all in all Equiformratings remains ahead on the surface.

Thus ends the final day, and a cracking final day for the Postdata oddsline. In fact, this reconciles the profit and loss position on this strategy over the whole 2 weeks, retrieving even the disasters of the first week, although as we will see in the next section, the position is somewhat reduced by commissions.

Table 14-17: P&L by Oddsline Strategy, December 1st 2007

Oddsline	date	Bets	wins	P&L	cum.P&L
Equiform	01/12/2007	67	8	1.25	776.16
Postdata	01/12/2007	23	4	289.99	55.75

Summary

So, it's clear from the last day's table that we made a profit. How do results look over the two weeks in total? Let's look by strategy, by running the query we have shown on most days displaying the number of bets per day, the number of wins, P&L per day, and cumulative P&L, for each strategy

Table 14-18: Equiformratings Oddsline Strategy

date	Bets	wins	P&L	cum.P&L
18/11/2007	34	4	71	71
19/11/2007	43	8	100.16	171.16
20/11/2007	37	3	-83.35	87.81
21/11/2007	62	8	-13.95	73.86
22/11/2007	51	5	-41.5	32.36
23/11/2007	42	5	-43	-10.64
24/11/2007	57	7	117.75	107.11
25/11/2007	49	9	342.5	449.61
26/11/2007	48	8	265.5	715.11
27/11/2007	40	5	8.04	723.15
28/11/2007	63	4	-224.75	498.4
29/11/2007	39	6	162	660.4
30/11/2007	38	7	114.51	774.91
01/12/2007	67	8	1.25	776.16

Table 14-19: Postdata Oddsline Strategy

date	bets	wins	P&L	Cum.P&L
18/11/2007	33	3	-18.3	-18.3
19/11/2007	33	2	-76.5	-94.8
20/11/2007	25	1	65	-29.8
21/11/2007	52	3	-138.69	-168.49
22/11/2007	51	4	-18.26	-186.75
23/11/2007	37	2	-110	-296.75
24/11/2007	44	5	26.5	-270.25
25/11/2007	20	0	-100	-370.25
26/11/2007	14	3	204.01	-166.24
27/11/2007	15	1	-56.5	-222.74
28/11/2007	16	1	-49	-271.74
29/11/2007	12	1	22.5	-249.24
30/11/2007	16	2	15	-234.24
01/12/2007	23	4	289.99	55.75

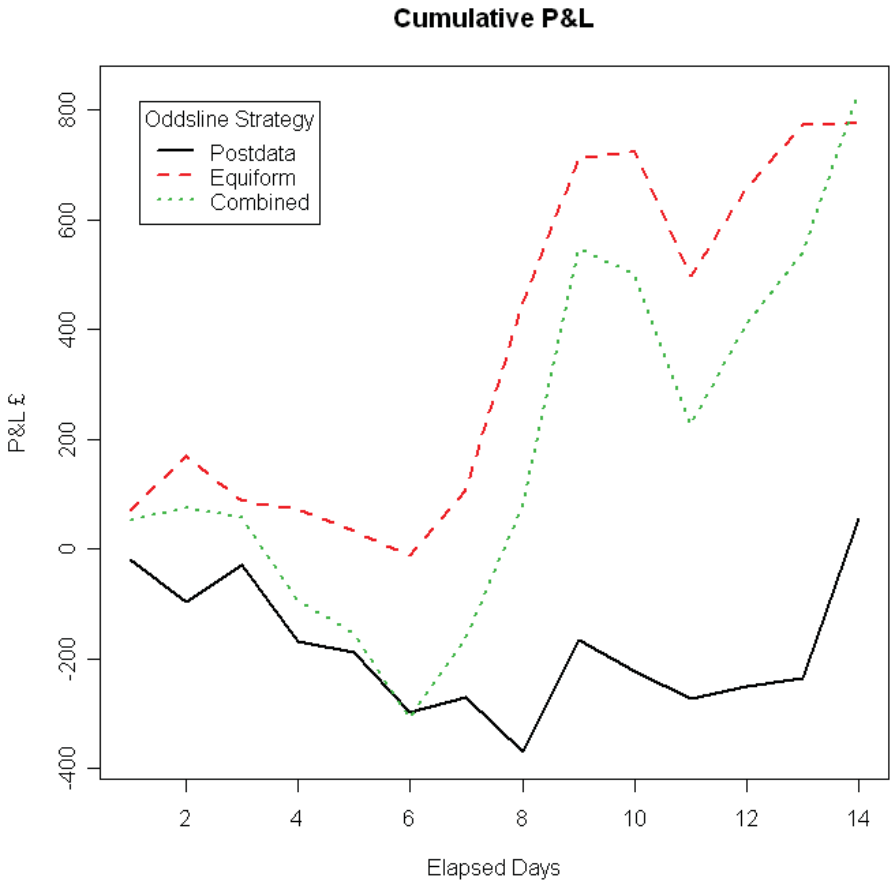


Figure 14-14: P&L over Production Run

Figure 14-14 shows the ongoing performance of the two strategies over the duration of the production run.

The combined strategies produced **£831.91** by the end of the fourteenth day. In the actual account, this total was reduced by commissions on all winners, at an average rate of approximately 4%, to create a realised profit over the period of **£661.80.** The Postdata strategy bet in all race types on days 1 – 7 inclusive, and, after our review at the end of the first week, on National Hunt handicaps for the second week. The first week was loss making, whilst the second week was profitable, breaking even for the strategy as a whole.

We successfully executed 1061 bets, at 5 minutes prior to each race, successfully taking the available to back price to a unit stake of £5, and we did this automatically over hundreds of races.

By automatically recording the success or otherwise of all bets, we have put in place review and data collection which can prove useful in testing and improving the strategy for the future. By following the bets interactively, we also can develop a sense of how the strategy behaves, and to cut down on surprises when we are not around to monitor it in the future.

Above all, we have made a profit – automatically.

Chapter 15: Next steps for the automatic bettor

In Chapter 14 we ran two strategies in production. The Equiformratings oddsline has been tested against historic results on an ongoing basis, and the odds are adjusted based on that testing. The Postdata oddsline was based upon a hypothesis, and had not been tested against historic results.

Implementing one strategy over the two weeks was clearly more risky than the other, given it had uncertain potential. We had to adjust the Postdata oddsline mid-flow, based on the little data we had, in an effort to improve its performance. The Postdata oddsline was used as an implementation example first and foremost, so the fact we managed to salvage a profit is a plus.

There were also other reasons for testing besides making a profit: We wanted to show:

- that the strategy was executed exactly as we had planned, incorporating the improvements from the live test,
- that the whole implementation from hardware infrastructure to software set up could be relied upon to run unattended in the long run
- that our assumptions about market conditions (i.e. ensuring that we could get on by taking the currently available back price to relatively small stakes – i.e. £5) were correct, and we could therefore continue with a simple implementation that did not require unmatched bets
- that key elements of the implementation, such as betting to an automatically generated events schedule, could be applied as a general mechanism to many different strategies.

In Chapter 12, we also referred to the possibility of creating a simulated test for any new strategy, provided that we had a reliable implementation in all other respects.

Testing Strategies without risk

When we began Part 3, we were lacking knowledge of how the example Postdata strategy operated in practice. Also, the implementation itself was untested for our first run in Chapter 12, where it turned out that some races were ignored, and others failed to match horse names correctly from the different input sources.

Having corrected these problems, we have a robust implementation, as we proved in the previous Chapter on the production run. We also have a much closer view on some of the strengths and weaknesses of the betting strategy. However, we do not necessarily have a robust betting angle.

Some areas looked promising and some weak, but our biggest problem is that without more results data than over a 14 day period it was difficult to be sure how to proceed, or how we could measure the long term accuracy of the oddsline.

This is a common problem with any new betting strategy, where we would like some confidence in how it will perform over a period of time, or to analyse "live" results, before committing capital.

The solution is to maintain the automation of the strategy as we have already set it up, but "turn off" the betting functionality in the program. This can be done simply by commenting out the part of the code relating to the **Placebets** call and adding a line to set any bet hash values to null as in:

```
my %bets;
```

Consequently, when the program is executed on its automatic schedule, all output will be produced as before, except for the fact that no call is made to execute bets and thus no Betfair identifier will be present. We have just had an automated non-bet! In this case, all the output showing market prices and so on will be consistent with the standard program report examples we have been showing (see for example *Figure 14-13*), but the final lines showing bets made will read as follows:

```
Bet top third overlays:
Selection:   3.45, Newbury, Kings Revenge, 20717542, 1117405
Bet placed: 0, B, £5.00,  15.5
Selection:   3.45, Newbury, Mon Michel, 20717542, 1233117
Bet placed: 0, B, £5.00,  23.0
```

In this case, we have a program report on our non-betting strategy but no simulated results to analyse, unless we write yet another program to extract all the lines beginning with **`Selection`** and then parse out the values and write them to a database. There's no point in doing that since we can write the selections to a database at the time we run the program.

Far more efficient then to create a database table, let's call it ***`simulated_bets`***, and simply write out the would-be bet details consisting of all elements that we want to record and analyse, not forgetting price and stake. (*Example 5-1* and *Example 8-1* provide code that can be adapted in terms of setting up a database table, with *Example 5-2* and others showing how to insert values to it within the context of a program).

Now we have an automated betting simulation but no results. So we need to reconcile winners and losers from our simulation using a results database, then calculate the profit and loss for the account.

To ensure everything is automated, even in the simulated test, we can use the *SmartForm Racing Database*, scheduling an automated query to access the database after results have been automatically updated. Our program will read in the table of simulated bets for that day's racing, one record per line. For each line, we save the race details (time, course and date) for the bet and use this information to present a query to the *SmartForm Racing Database,* retrieving the runners and finishing positions for that race. The runners are then matched to the name of the selection for the simulated bet, together with its winning position. This information is written back to the database table for simulated bets in order to update finishing position and profit and loss. We can calculate profit and loss as follows, given there is a finishing position available:

```
if ($finisPos == 1) {$pandl = $stake * ($decimal_odds - 1)} else {$pandl = (- $stake)}
```

If the horse is a non-runner (e.g. it is withdrawn after the bet has been placed), we record a value of zero. Whatever the source of results data used, careful attention must again be paid (as with the comparisons of oddsline runner names to Betfair runner names) to comparing runner names from different sources.

In this way we can continue to build a database of results and simulated P&L for any strategy for as long as required, until going live or abandoning or modifying the strategy.

Betfair Account Management

We refer here to setting up a Betfair account with some common sense guidelines for optimal use of automated betting. Account Management functions are also available through the API, as listed in Appendix 3, so can be automated as required.

Stop loss limits:

On every Betfair account there is a weekly, monthly, yearly loss limit that can be set within the profile of each user (available in the My Account tab. When experimenting with automating betting for the first time, it can be useful as an absolute failsafe.

However, it is better to think of this as a failsafe rather than a management tool, since limit changes take up to 7 days to come into effect, and the loss limits are inflexible.

Ideally, any use of stop limits in this way will therefore be replaced by the bettor's own programs to monitor strategies and review profits and losses.

More than one account

If the bettor is running automated strategies and continuing to bet interactively on Betfair, it is generally advisable to do so from separate accounts. Otherwise, there can be room for error and confusion if dealing in the same markets. For example, we can think of attempting to interactively trade the favourite in a market where a position has already been taken on the same horse by an automated program. We can do this, of course, but it adds unnecessary confusion and difficulty to the process if we want to maintain the integrity of the "long" position of the automatic bet.

The downside of using separate accounts is not benefiting from reductions in commission rates due to pooled activity. A solution is a Betfair master account with sub-accounts, where the sub-accounts are linked for commission purposes, although this may not be available to all since conditions for issuance are subject to account activity.

In the absence of a master account, it is feasible to manage one account for several automated strategies, with no downside to executing and calculating P&L on each strategy automatically - this being the scenario we support for Chapter 11. But one separate account for interactive betting is still preferable.

Don't Forget to Pay Yourself

In looking at the important factors of testing strategies first, mitigating risk, using stop loss limits, and thinking of long term results over short term volatility, let's not lose sight of the wood for the trees.

Although no *interactive* Betfair facility exists for automatically paying yourself (as it does for stop losses), taking regular profits is an equally important aspect of account management.

A useful strategy is therefore to set achievable account balance targets for an automated strategy and withdraw excess capital once those targets are reached.

This can also be automated using relevant elements of the general framework covered in Part 2, along with an additional API call to **withdrawToPaymentCard**. A weekly check with a scheduled **`cron`** job (see Chapter 7) to check account funds (see the section on *Automating staking plans using a betting bank* in Chapter 9 for such an example using the **GetAccountFunds** service) will give us the account balance.

A simple program (scheduled to run at the same time as checking funds or as part of the same program) can then subtract the target balance (which can be "hard coded" or based on an algorithm related to the strategy) from the available funds and will tell us if we are making the desired profit margin required for withdrawal. If we are - in other words, if there is an excess balance indicated by a positive number - we specify the result as the excess amount to be withdrawn from the account, then can use the API call to transfer that amount to an existing payment card.

Avoiding Cognitive Bias

Cognitive bias is a term coined in behavioural finance relating to becoming emotionally attached to, or responding emotionally to, an investment or trade in a way that can be detrimental to obtaining the best outcome. Since betting or trading on horseracing and other sports can be almost no different to investing or trading in financial markets, it is a bias which is also apparent in many betting decisions.

Nevertheless, it may seem strange to include a section on betting psychology in the context of automated betting - especially when we have discussed how an automated strategy can help to bring consistency to the betting process and remove the damage that can be caused to generally profitable strategies by emotional or manual betting decisions (e.g. a bettor staking far more than usual if on a "winning streak", or failing to apply methods consistently and so on).

Whilst automation can avoid these problems, since the bettor is still in ultimate control of an automated strategy, he or she can pull programs out or put programs into production at whim. In this sense, an automated strategy is only as good as the bettor in charge of it.

Betting on an automated basis can be completely different to betting interactively, so it is as well to be prepared for what to expect and how you will react. Especially if the strategy is followed too intensely, it can cause impatience, doubt in the methods being used and premature interference in the strategy just as easily as betting interactively can - even when the automated strategy is showing a profit.

There are many reasons or situations in which detrimental interference – meaning tampering with stakes, altering decision making rules and so on - can happen. Here are a few:

- Second guessing the automated strategy – the bettor is toying with a new strategy and picks a winner in a race, does not back it, and watches the automated strategy bet a loser.
- The strategy goes sideways or downwards in terms of profits over 2 months or more, so the bettor pulls the plug or reduces stakes, only to watch profits go up afterwards – ignoring the fact there was a similar 2-3 month downturn when backtesting the strategy.
- The bettor sees a small profit and withdraws it early, significantly reducing the bank and unit stakes - thereby limiting future gains.
- The bettor interferes with or increases stakes to try and recoup losses - increasing the risk of “gambler’s ruin”.

These are the same types of betting decisions that can damage the profitability of sound manual strategies, and there is no reason, just because a strategy is automated, that the bettor cannot interfere with the process in a similar way - although automation of a fundamentally sound strategy can avoid all these issues if left to its work.

The objective in automatic betting is often to find an edge which has been proven over time, test it, implement it, and watch it accumulate.

So, in order to let an automated strategy run its course, it can be helpful to think of the interaction with an automated strategy entirely differently to the way one would normally bet in a traditional process. That is, that the bettor is betting on the strategy itself – which is a long term play - rather than any individual bet.

By the same token, the bettor should beware of, or at least have an advance plan in place, for implementing any strategy which has not been extensively tested. Otherwise there is likely to be less confidence about how it will perform, and the natural reaction will be to try and “do” something about any adverse results, or to take profits early.

The Automatic Betting Portfolio

In this book we have investigated one of the more novel ways of going about the betting process, which brings a new dimension to the activity itself.

Automatic betting provides one answer to betting, rather than *the* answer. There is no reason why you cannot mix and match whatever strategies and betting media are at your disposal, according to what makes most sense given your strategy or approach, or the possibilities offered by the media available.

However, to come back to the statements we made at the beginning of the book, the strategies bettors have for predicting outcomes in sports betting often rely on data. This could be any type of form data, third party data or market data. It could be repurposed for use in a betting system or to create a predictive model. It could be static data or updated according to new inputs, where we might suppose that automation has no place (such as taking account of changes in the going in horseracing, or paddock reports that are accessible by programs online). The reasons bettors have for making decisions are often measurable or rule based – "if this, then that" - lending themselves well to programmation. Many factors and therefore strategies can change based on new observations or new opportunities – such as in-running markets on the betting exchanges – but does that mean automation is not possible for these new markets?

We have explored or touched on many of these ideas within the context of the book without providing full code for all – but, if in any doubt about the potential, take a look at almost any book about betting on horseracing and you will see there is no shortage of interesting angles or strategies that we can consider automating, in addition to the strategies that each existing bettor will hold high (and of course, if we extend that to the domain of trading, then the possibilities increase yet again).

The interesting question for the automatic bettor is how to define the process, to ask what data elements would be needed to automate a strategy, what decision process would a program need to implement in order to automate the strategy and, perhaps before contemplating any of the above, under what conditions, if any, it could be profitable.

Bibliography

Automatic Exchange Betting touches on many different subject areas including horseracing, sports betting, probability, exchange markets, IT and programming. This list concentrates on providing an overview to some of the most useful starting points, which themselves provide excellent references to further material.

Online Sources

www.betwise.co.uk	The author's site containing example code, data and other supporting materials for *Automatic Exchange Betting*.
www.betfair.com	The world's largest betting exchange. In particular, note the links to the Betfair Developers' Program and technical documentation, especially the *API Sporting Exchange Reference Manual*.
www.racingpost.co.uk	The online version of the newspaper widely seen as the best free interactive online form, ratings, and statistics service for UK and Irish horseracing.
www.sportinglife.com	Another excellent site for online form, results and news for a number of sports.
www.equiformratings.co.uk	Private horse racing ratings service used to demonstrate oddsline strategy automation in Part 3.
http://learn.perl.org	A useful starting point for those new to Perl, with links to other resources such as the Perl FAQ, Perl documentation, *www.cpan.org* and books.
www.ubuntu.com	Community, download and support for most popular Linux distribution, (recommended operating system for running example code).
www.mysql.com	Community, download and support for the most popular open source database (recommended database for running example code)

Further Reading:

The betting process:

Electronic information and online exchanges may have revolutionised the way in which people bet but the fundamentals remain the same. For an appreciation of the underlying process in betting on horseracing outcomes, *Betting for a Living* can be thought of from the perspective of the backer and *The Art of Legging* from the perspective of the layer:

Betting For A Living
Nick Mordin
Aesculus Press Limited; Second edition, 2002

The Art of Legging
Charles Sidney
Maxline, 1976

Systematic betting on horseracing

These references look at analysing form to uncover profitable angles and approaches to developing profitable betting systems on a quantitative basis.

Winning Without Thinking
Nick Mordin
Aesculus Press Limited, 2002

Against The Odds
David-Lee Priest
Raceform Ltd, Second Edition, 2005

Forecasting Methods for Horseracing
Peter May
High Stakes Publishing, 2002

Probability and Games of Chance

Taking Chances is a layman's guide to probability, including a practical explanation of how to implement the Kelly staking strategy. *The Theory of Gambling and Statistical Logic* presupposes more advanced mathematics for its explanations, including a large number of interesting further references.

Taking Chances
John Haigh
Oxford University Press, 2003

The Theory of Gambling and Statistical Logic
Richard A. Epstein
Academic Press, 1995

Programming - Perl

Some references for learning the language and specific aspects of Perl that are used in accessing the Betfair API, such as Perl and XML and the LWP library, as well as tips and tricks for general usage.

Learning Perl, Fourth Edition
Randal L. Schwartz, Tom Phoenix, Brian D Foy
O'Reilly, 2005

Perl and LWP
Sean M. Burke
O'Reilly, 2002

Perl and XML
Erik T. Ray, Jason McIntosh
O'Reilly, 2002

Perl Cookbook, Second Edition
By Tom Christiansen, Nathan Torkington
O'Reilly, 2003

Spidering Hacks 100 Industrial-Strength Tips & Tools
By Kevin Hemenway, Tara Calishain
O'Reilly, 2003

Linux and Ubuntu

Beginning Ubuntu Linux is a great start for those with no experience of Linux and includes a bootable CD, but for the more adventurous, *Running Linux* also provides an overview of Linux as a complete OS, as well as pros and cons of different distributions.

Beginning Ubuntu Linux, Second Edition (Beginning from Novice to Professional)
Kier Thomas
Includes Ubuntu DVD-ROM
Apress, 2007

Running Linux, Fifth Edition
Matthias Kalle Dalheimer, Matt Welsh
O'Reilly, 2005

MySQL

The canonical third party guide to MySQL below, followed by the example based alternative, or complement, by the same author.

MySQL Third Edition
Paul DuBois
Sams, 2005

MySQL Cookbook
Paul DuBois
O'Reilly, 2007

Appendix 1: Software configuration for example code

This Appendix provides notes on setting up your computer system to run the example code that is described in the book and available for download online at *www.betwise.co.uk*. The notes are for general guidance only and not intended as a substitute for more comprehensive instructions or reference books on installing and configuring operating systems or other software. For those with no prior experience in Perl, MySQL or Linux, the relevant references listed in the Bibliography are an indispensable prerequisite and accompanying guide; it is also worth reviewing *www.betwise.co.uk* for alternative solutions.

First some context: Each Chapter in Part 2 provides example Perl scripts which illustrate automating different stages of the betting process, with many scripts themselves using one or more Perl functions from a library of functions that call API services (documented in Appendix 2). The examples are provided within the context of an overall software framework for automation that relies on scheduling facilities and other shell commands available within a Linux or other UNIX based operating system, coupled with data persistence that is achieved using a MySQL database.

Other programming languages, operating systems and databases than those used by the examples can of course be used to achieve the same objectives. Thus the code examples can be seen as "pseudo code" for illustration purposes, or can be used or adapted "as is" for experimenting with the API or creating automate betting strategies. However, to run the examples without any adaptation, a machine running Linux, Perl, certain Perl modules and MySQL installed is required. In the case of a user who does not already have a preference or experience with this type of environment, Ubuntu is strongly recommended as the choice of operating system, since it can be pre-installed in many new PCs, and provides the greatest ease of use for installation of dependent software and modules required (it is also the most popular Linux distribution).

Furthermore, Ubuntu can be installed in dual boot mode alongside other Operating Systems (OSs) such as Windows, or as a virtual machine within a Windows OS for those who do not wish to switch from Windows and/or do not have more than one machine available. Ubuntu is therefore the default environment described for running the example code, as covered more fully in "Notes for installing Ubuntu", below.
For Mac OS X users the examples can also be run "as is" within the Terminal application, which provides access to the underlying UNIX based OS; further notes on setting up Mac OS X for Perl and MySQL are provided in the subsection "Notes for Installing to Mac OS X". For users of other Linux distributions, the examples can be run more or less as with Ubuntu, with further notes in the subsection "Notes for other Linux distributions". Finally there are notes on configuring the code to run on Windows systems in the subsection "Notes for Windows".

Notes for installing Ubuntu

This section assumes you wish to run the code examples on a dedicated machine where the operating system is installed from scratch. Note that the following subsection deals with installation of Perl and MySQL, which applies to existing Ubuntu installations.

Unless your PC has come with Ubuntu installed, you will need to download the required ISO image or order a free CD for installation by following the links for downloads at *www.ubuntu.com*. Ubuntu CDs are available from many other channels, including shipping free with books on the Operating System.

In terms of which version to install, the latest "LTS" release (i.e. 'long term support' release) is usually a good idea – an overview of the support policy for different releases is described at the Ubuntu website.

Installing Ubuntu is a matter of booting the CD or clicking the downloaded ISO image and following the prompts. When installing, the user has the option to overwrite the hard drive to make Ubuntu the only OS on the install machine, or partition the hard drive to share with an existing operating system, such as with an existing Windows machine. We outline options for preserving Windows and installing Ubuntu alongside in the "Notes for running code on Windows".

One of the advantages and design goals with the Ubuntu Linux distribution (and its variants Kubuntu etc) is the relative ease of installation. Nonetheless it's always useful to have references at hand, and fuller details of the installation process, including configuring hardware and network options, are covered online at *www.ubuntu.com* under the links to support and then documentation as well as other sections of the website. Additionally, there are plenty of other good reference books and guides available some of which are included in the Bibliography.

Installing Perl and requisite Perl modules in Ubuntu

Whether you have installed Ubuntu from scratch as the only OS on your machine, are already using Ubuntu, or are using a dual-boot or virtual machine with Ubuntu as one of the operating systems available, the next step will be to ensure that the requisite versions of Perl and dependent modules, as well as MySQL, are also installed.
One of the nicer aspects of using Ubuntu for setting up an open source software environment for the first time is the ease of installing other required open source software, since this can all be done from within the Ubuntu GUI, with all dependencies taken care of for the user.

The facility to use from the GUI is Ubuntu's *Synaptic Package Manager* which is accessed from the Home Menu via *System -> Administration -> Synaptic Package Manager*. From the command line, which is the method we will describe, the *apt-get* facility can be used to install additional software.

N.B. To access the command line from the GUI, select *Applications -> Accessories -> Terminal.* Or, to access the command line from a virtual terminal, press Ctrl + Alt + Function Key – generally F1 through to F6, with F7 returning the user to the GUI - then type in your username and password at the prompts.

First, we need Perl, which is usually installed by default in most Linux distributions, including Ubuntu. To check which version of Perl is installed go to the command line and type perl –v, then press Enter. The version of Perl – 5.8 or higher will be fine - will be displayed, together with details of the open source licence.

Now we need to add some specific modules to the Perl installation that will be used by the example programs.

All installs should be done as the root (or "admin") user, which can be temporarily achieved by prefixing any install command with **sudo.**

The generic (i.e. non-OS specific) process of installing modules to Perl is via command line access to **cpan**, (Comprehensive Perl Archive Network), installing each module required. This can be done in Ubuntu also, (although has not tended to work as seamlessly "out of the box" as with other Linux distributions). On the other hand, the advantage of Ubuntu is that many associated Perl modules are available within bundled packages that can be easily installed via the **apt-get install** procedure, which is what we will show here.

First, whether installing via **cpan** or the bundled packages, it is as well to make sure the right kit is available with the following, hitting enter after the command:

```
sudo apt-get install build-essential
```

Then wait until the prompt is returned and install the following four packages in the same way:

```
sudo apt-get install libxml-perl
sudo apt-get install libxml-xpath-perl
sudo apt-get install libdbi-perl
sudo apt-get install libcrypt-ssleay-perl
```

Provided all builds are successful, this takes care of the install for the example Perl code to work on Ubuntu *Hardy Heron*, version 8.04. It should also take care of most dependencies on other Ubuntu versions and ensure that the example code works fine (provided that the scripts themselves are set up correctly, as discussed in the next subsection). However, it's always possible that some dependent or required modules are found that are not handled in the particular set up you have. In such cases, the error messages are usually self-explanatory – stating that certain modules that were expected are missing. In this case, a trip to the CPAN archive, as with:

```
sudo perl –MCPAN –e shell
```

can help to locate and find the relevant missing modules in most set-ups. However, in Ubuntu, the trick is to minimize the effort by looking for the appropriate bundled package containing the module in question, and then use the Ubuntu install mechanism of **sudo apt-get install** for the relevant package. In this respect, the Ubuntu website is a useful resource where you can find the required Perl modules within various bundled packages.

Installing MySQL in Ubuntu

As with the instructions for Perl above, a convenient way to install MySQL – in the case of Hardy Heron, 8.04, this is MySQL5 – is simply to access a command line and use **`apt-get`** to install the package for MySQL bundled with the distribution:

```
sudo apt-get install mysql-server
```

During the course of the installation you should be asked for a password for the root user.

Assuming all runs successfully, you will be in a position to set up a database and users with permissions for accessing the database. For the purpose of the examples in the book, the database used is called **autodb.** We cover creating the **autodb** database together with a default user with appropriate permissions (for the purposes of running the examples at least) in the last section, *Configuration for Example Scripts, Perl API Functions and AutoDB Database.*

Notes for Installing to other Linux distributions

Those who have already used Linux will most likely have their own preferred distribution. There is nothing complicated about the example code that makes configuration within any other Linux distribution more complex than Ubuntu (and has been run on several, including SuSE and Red Hat) but will rely on the way in which Perl, Perl modules and MySQL are handled within those other distributions - or indeed the way in which the user already has them set up if familiar with the environment.

The general advice would therefore be to consult the documentation associated with the specific distribution for installing MySQL 5, Perl and Perl modules. No doubt Perl will already be present and can be verified as for the notes on Ubuntu above. Most Perl references will also contain instructions on connecting to CPAN and building Perl modules from there, irrespective of the bundled packages within any distribution.

The Perl packages and modules which may be required depending on the base installation, are:

```
libwww-perl
DBI
XML::Simple
XML::Parser
XML::XPath
Crypt::SSLeay
```

Any dependencies not listed, depending on your installation, will be flagged as needed, and can be resolved by visiting CPAN.

Again, binary installs for MySQL will most likely be available, but can always be downloaded together with installation and configuration documentation at *www.mysql.com.*

Notes for Installing to Mac OS X

At the heart of the OS X line of modern Apple Mac machines is a UNIX OS based on the FreeBSD distribution, meaning that the software environment can be configured mostly as for any other Linux distribution.

The command line can be accessed via the Terminal window (a quick "perl –v" at the command line will confirm Perl is again installed as standard), and software dependencies with Perl, Perl modules and MySQL can be verified, installed and configured exactly as for the notes on installing to other Linux distributions, as above.

There is also the possibility to install Ubuntu as a dual boot system or virtual machine on a Mac OS X if required, although this is not really necessary for running the examples.

Notes for installing to Windows

Ironically the first point to consider about running the examples on Windows is not to use Windows at all. Since the code is set up for a UNIX based environment, it requires some adaptation before running natively on Windows, so it can be easier to remain within a Linux environment. Moreover, there are many options now for running any Linux distribution (including Ubuntu) on the same machine as an existing Windows installation, without affecting the existing OS and filesystem. – from dual-booting the machine by partitioning the hard disk, to running a virtual machine, or running Cygwin within a Windows system. Some of these set ups can be complex (further details can be found in Ubuntu references in the Bibliography), but setting up a dual boot system is an "out of the box" option with Ubuntu, when installing the CD, so is to be recommended – with the usual caveats applying to making backups before touching a system you want to preserve. Further notes on creating a dual boot system are available within a number of the references already made, including the Ubuntu website.

Once the dual boot system is created, the rest of the configuration required to set up the environment for running examples with Perl and MySQL is as described previously under *Notes for Installing Ubuntu*.

However, if wishing to run the examples using the Windows OS, further adaptation of the code is required, although the good news is that Perl and MySQL run fine on Windows systems.

First, to download the recognised standard Windows distribution of Perl (i.e. Activestate) go to *www.activestate.com/downloads*

Once installed, there is much useful documentation available within the Windows Programs menu where Perl will install by default. Note in particular the sections about running Perl in a Windows environment. With regard to installing the prerequisite modules, use the Perl Package Manager, also available from the Program menu. At the command prompt, search for and then install the prerequisite Perl modules listed under *Notes for Installing on Other Linux Distributions*.

MySQL 5 can be installed and configured by following the appropriate download links for your Windows system at *www.mysql.com*, then following the install instructions.

The example scripts themselves require adaptation for execution within Windows (for example, with regard to handling path names which are treated differently between Windows and UNIX systems). Automating the scripts to execute an entire betting strategy in the way described for Linux will also require some reading (or prior knowledge) on executing scripts, permissions and scheduling in a Windows environment. Windows does have command line scheduling facilities (as described in Chapter 7) which are also accessible via the GUI, so in principal an automated system can be created leveraging scripted languages and tools within the OS (as with the Linux examples), but generally this can be trickier than setting up UNIX type systems in the same way, so it is a case of *caveat emptor*.

Configuration for Example Scripts, Betfair API Functions and Example Database

This section provides some guidelines for running the example code, assuming the required software – which is to say the operating system, Perl and MySQL installation – has first been correctly installed and configured as per the notes in the previous sections.

General

Running the examples relies on interaction between the following:

- Perl scripts or programs for automating specific parts of the betting process
- A library of Perl functions for accessing the Betfair API
- A database to achieve data persistence between programs
- Leveraging scheduling facilities within the OS.

Since all this interaction is automated, a couple of general principals in running the example set up (i.e. so that it does not fall down between components) are critical:

- Ensuring appropriate permissions from one script or executable to another
- Ensuring installation of programs, function libraries and commands where other programs, function libraries and commands expect to find them

The above is common sense, but can be where most trouble occurs. Specifically, all program names or commands must be executable by the operating system. A blanket approach to grant all users all permissions would be logging in as **`root`** and making any program executable by typing **`chmod ugoa+rwx program_name`**, but see the **`chmod man`** page for more granular control (especially if you are not the only user on your system!). The operating system must of course know how to run any program or command (i.e. with whatever relevant commands or interpreters), and where to find dependent interpreters commands, files or scripts, either through an explicit path that is specified, or because the global **`$PATH`** variable has been set appropriately.

Paths

All system paths in the examples are hard coded with a default to the **/home** directory of a user **/aeb**. The assumption is that all paths should be changed to a directory structure reflecting the actual user's system or path of choice.

Most example programs themselves call functions, or subroutines, to access particular API services. The library, or package of functions, that defines access to API services, including the arguments used for each call and the data structures returned, is called **BetfairAPI6Examples.pm.** The library is listed in Appendix 2.

The default path used within the example programs to the library of Perl functions is the directory ***/lib*** available from the home directory of the user. So the access path (for user **aeb** is ***/home/aeb/lib/BetfairAPI6Examples.pm.***

Again this should be changed (in accordance with the Perl documentation for ensuring program by program access to user defined libraries of functions) to any system path preferred by the actual user.

It is assumed that all other examples and data are accessed from the user's home directory ***/home/aeb.*** Paths should be amended as required but should of course remain consistent with each other.

Creating the Example autodb Database

The **autodb** database is a set of simple database tables to store and access data (such as Betfair race IDs) required by the example programs.

Set up for each database table in the `autodb` database is explained whenever a new table is required by examples in the main text, but setting up any table first requires that the database itself exists. Creating this database is described below.

First, ensure that MySQL is installed as per the previous notes in this Appendix and type the following to log into the mysql monitor at the command prompt (> below):

```
>mysql -u root -p
```

You should now enter your password, provided that one was set up during the installation of MySQL

```
       >Enter password:
```

Whereupon you will successfully be logged in

```
       Welcome to the MySQL monitor.  Commands end with ; or \g.
```

Next, set up the database as follows:

```
>CREATE DATABASE autodb;
```

Then, grant access to the user who will be setting up tables and executing programs automatically – probably the same as the main user on a single user system.

```
>GRANT ALL ON autodb.* TO 'your_account_name'@'localhost';
```

Log off from the MySQL monitor to return to the shell prompt:

```
>exit;
```

From now on you can log yourself in by the specified username to interact with the database, or specify the same user credentials within the context of a Perl script. This will enable programmatic database operations using the DBI module, which can be scheduled automatically, as discussed in the program examples.

Example Crontab Entries for Automation of Oddsline Betting

In Chapter 7, we discussed that scheduling the example programs using the **nix* (Unix or Linux) crontab facility can facilitate the automation of most if not all betting strategies. Indeed, this is the way in which the example programs for the oddsline betting strategy begun in Chapter 6 and followed through towards live betting in Part 3 are automated.

Below we describe a typical structure including notes for crontab entries (for the example programs described in Part 2 of the book) which can be used as the basis for automating the example strategies in Part 3.

```
#assume a crontab file
# m h  dom mon dow   command

#replace /home/aeb/  with your system path and name programs to suit.
#Below program names relate to those used in the book, with example numbers also
referenced

00 08 * * * /home/aeb/get_betfair_races.pl          #Example 5.2

#Next, generate an oddsline and make it available in a consistently formatted flat file.
#This step is likely to include extra programs depending on the oddsline method - eg. if
#applying models to your own database, screen scraping, or converting third party
#ratings, all of which are automated at this point for the production runs shown on the 2
#oddsline methods used in Part 3.

05 08 * * * /home/aeb/format_oddsline.pl            #Example 6.1

#Next, generate a flat file of events for the day and make it available to all programs

10 08 * * * /home/aeb/generate_events_schedule.pl   #Example 7.1

#Next, schedule the strategy to run 5 minutes prior to each event based on event times

15 08 * * * /home/aeb/write_schedule.pl             #Example 7.2

#The above executes bet_oddsline_strategy 5 mins before each race, which contains 3
#programs to run consecutively for each race as follows:
#/home/aeb/get_market_prices.pl                     #Example 8.2
#/home/aeb/bet_formation.pl                         #Example 9.1
#/home/aeb/execute_bets.pl                          #Example 10.1

#N.b. In chapter 13 we discuss combining these 3 programs to create one shorter program
```

```
#e.g. oddsline_overlays.pl for production runs.   This single program can still be
#included in bet_oddsline_strategy (instead of the 3 above), so Example 7.2 is the same

#Now the programs will bet automatically until the end of the day's racing.

#The next entries in crontab record the performance of the strategy for the day
#Based on the assumption that British racing is over and bets settled by 10 pm every day.

00 10 * * * /home/aeb/get_betting_performance.pl    #Example 11.2

#Next, reset files ready for the next day's betting, ensuring that no manual interference
#is needed to run the betting strategy the next day

#First, archive and/or remove the file betting_report from the production directory

04 10 * * * rm /home/aeb/betting_report

#Next, archive and/or remove the file daily_bets from the production directory

05 10 * * * rm /home/aeb/daily_bets
```

Appendix 2: Example Perl library for Accessing Betfair API services

This Appendix lists the example Perl library for accessing and returning values from Betfair API services. The library is introduced in Chapter 4 and subsequently referred to by other examples. This example code is available for download at *www.betwise.co.uk*, and maintained to reflect any changes in the Betfair API as time permits. Instructions on setting up the library for access by other scripts are provided in Appendix 1.

```
package BetfairAPI6Examples;
use LWP::UserAgent;
use LWP::Debug; # qw(+trace +debug +conns);
use HTTP::Request;
use HTTP::Cookies;
use Data::Dumper;
use XML::Simple;
use XML::XPath;
use strict;

#-------------------------------------------------------------------------------------#
# This code is Copyright (c) Colin Magee, 2004-2008.  All rights reserved.            #
# The code from this library is provided under the terms of the Artistic License 2.0  #
# Code download including full licence terms at http://www.betwise.co.uk/aeb/code     #
#-------------------------------------------------------------------------------------#

#       This library of example Perl subroutines accesses services from the Betfair API,
#       version 6.

#       Full descriptions of the parameters available for return by each service can be
#       found in the official API documentation from the Betfair Developers Program at
#       www.betfair.com.

#       All subroutines are for operation on the UK global exchange, as per the examples
#       in Automatic Exchange Betting.  Adapting these subroutines for Australian events
#       is discussed in Chapter 4 of the book.

#       History and acknowledgements:
#       2004:  Betfair API3:  Perl subroutines were originally written by the author with
#       BDP support to access Betfair API3 using the SOAP::Lite module for Perl
#       2006:  Betfair API4:  API4 changed from RPC to doc/literal, unsupported in
#       SOAP::Lite.  Thanks to code posted at the Betfair Developer Forum, this library of
#       subroutines was rewritten for API4
#       2007: Betfair API5:  API5 added the Australian exchange.  Example subroutines were
#       adapted to handle different URL endpoints.
#       2008: Betfair API6: Betfair added new services and existing services were adapted
#       to support Betfair SP and Persistence bets, as reflected in the current examples.


require Exporter;
our @ISA = qw(Exporter);
our @EXPORT=
qw(login get_market_prices_compressed get_markets get_events place_bet
       get_detailed_market_depth get_market_traded_volume get_bet get_mubets
       get_market_pandl
update_bet cancel_bet get_account_statement get_account_funds);
our $VERSION= 1.0;
our $cookie_jar = HTTP::Cookies->new(hide_cookie2 => 1);

sub login
       {
  my ($username,$password,$productId)=@_;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <SOAP-ENV:Body>
```

```
                <m:login xmlns:m="http://www.betfair.com/publicapi/v3/BFGlobalService/">
                        <m:request>
                                <password>'.$password.'</password>
                                <productId>'.$productId.'</productId>
                                <username>'.$username.'</username>
                                <vendorSoftwareId>0</vendorSoftwareId>
                                <locationId>0</locationId>
                        </m:request>
                </m:login>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/global/v3/BFGlobalService');
$request->header(SOAPAction => '"
https://api.betfair.com/global/v3/BFGlobalService
"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);

my $ref;

eval { $ref = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "login failed to retrieve valid
        XML"};

my %login_hash = ();
$login_hash{sessionToken}     =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'header'}{'sessionToken'}{'content'};
$login_hash{headererrorCode}  =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'header'}{'errorCode'}{'content'};
$login_hash{errorCode}        =
$ref->{'soap:Body'}{'n:loginResponse'}{'n:Result'}{'errorCode'}{'content'};
        return %login_hash;
        }

sub get_markets
        {
  my ($sessionToken,$marketId)=@_;
        my %market_hash= ();
                my %names_hash = ();
                my $runnerId = ();
                my $runner_name = ();

  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getMarket
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <marketId>'.$marketId.'</marketId>
                        </m:request>
                </m:getMarket>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
```

```
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "get_markets call failed"};
#print Dumper(\$result);


  my $response=\%{$result->{'soap:Body'}{'n:getMarketResponse'}{'n:Result'}};
        $market_hash{'errorCode'}          =
        $response->{'errorCode'}{'content'};
        $market_hash{'timeStamp'}          =
        $response->{'header'}->{'timestamp'}{'content'};
        $market_hash{'sessionToken'}       =
        $response->{'header'}->{'sessionToken'}{'content'};
        $market_hash{marketTime}      =
        $response->{'market'}{'marketTime'}{'content'};
        $market_hash{BSP}             =
        $response->{'market'}{'bspMarket'}{'content'};
        $market_hash{canTurnIn-play}  =
        $response->{'market'}{'canTurnIn-play'}{'content'};
        $market_hash{marketType}      =
        $response->{'market'}{'marketType'}{'content'};
        $market_hash{marketSuspendTime}       =
        $response->{'market'}{'marketSuspendTime'}{'content'};
        $market_hash{numberOfWinners} =
        $response->{'market'}{'numberOfWinners'}{'content'};
        $market_hash{eventTypeId}     =
        $response->{'market'}{'eventTypeId'}{'content'};
        $market_hash{countryISO3}     =
        $response->{'market'}{'countryISO3'}{'content'};
        $market_hash{timezone}        =
        $response->{'market'}{'timezone'}{'content'};
        $market_hash{discountAllowed} =
        $response->{'market'}{'discountAllowed'}{'content'};
        $market_hash{menuPath}        =
        $response->{'market'}{'menuPath'}{'content'};
        $market_hash{name}            =
        $response->{'market'}{'name'}{'content'};
        $market_hash{marketDisplayTime}       =
        $response->{'market'}{'marketDisplayTime'}{'content'};
        $market_hash{marketStatus}    =
        $response->{'market'}{'marketStatus'}{'content'};
        $market_hash{marketBaseRate}  =
        $response->{'market'}{'marketBaseRate'}{'content'};
        $market_hash{parentEventId}   =
        $response->{'market'}{'parentEventId'}{'content'};
        $market_hash{runnersMayBeAdded}       =
        $response->{'market'}{'runnersMayBeAdded'}{'content'};
        $market_hash{marketDescription}       =
        $response->{'market'}{'marketDescription'}{'content'};
        $market_hash{lastRefresh}     =
        $response->{'market'}{'lastRefresh'}{'content'};
        foreach my $s (@{$response->{'market'}->{'runners'}{'n2:Runner'}})
                {
                $market_hash{runners}{$s->{selectionId}{'content'}}{asianLineId}
                =$s->{asianLineId}{'content'};
                $market_hash{runners}{$s->{selectionId}{'content'}}{name}   =
                $s->{name}{'content'};
                $market_hash{runners}{$s->{selectionId}{'content'}}{handicap}
                =$s->{handicap}{'content'};
#To extract most frequently used hashes, create 2 arrays and take a reference to each

                $runnerId = $s->{selectionId}{'content'};
                $runner_name = $s->{name}{'content'};
                $names_hash{$runner_name} = $runnerId;
                }
        my @hashes = (\%names_hash, \%market_hash);
        return @hashes;
```

```
        #return %names_hash;
        }

sub get_events
        {
  my ($sessionToken,$eventParentId)=@_;
        my %events_hash;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getEvents
        xmlns:m="http://www.betfair.com/publicapi/v3/BFGlobalService/">
                        <m:request>
                                <header>
                                        <clientStamp>7687789</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <eventParentId>'.$eventParentId.'</eventParentId>
                        </m:request>
                </m:getEvents>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';

my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/global/v3/BFGlobalService');
$request->header(SOAPAction => '"https://api.betfair.com/global/v3/BFGlobalService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "get_events call failed"};
#print Dumper(\$result);
 my $response=\%{$result->{'soap:Body'}{'n:getEventsResponse'}{'n:Result'}};
# print Dumper(\$response);
        $events_hash{'errorCode'}             =
        $response->{'errorCode'}{'content'};
        $events_hash{'timeStamp'}             =
        $response->{'header'}->{'timestamp'}{'content'};
        $events_hash{'sessionToken'}          =
        $response->{'header'}->{'sessionToken'}{'content'};
        my $type=substr($response->{'marketItems'}{'n2:MarketSummary'},0,4);
        if ( $type eq "HASH" )
                {
                my $marketId=
                $response->{'marketItems'}{'n2:MarketSummary'}{'marketId'}{'content'};
                $events_hash{markets}{$marketId}{timezone}   =
                $response->{'marketItems'}{'n2:MarketSummary'}{timezone}{'content'};
                $events_hash{markets}{$marketId}{menuLevel}  =
                $response->{'marketItems'}{'n2:MarketSummary'}{menuLevel}{'content'};
                $events_hash{markets}{$marketId}{marketName} =
                $response->{'marketItems'}{'n2:MarketSummary'}{marketName}{'content'};
                $events_hash{markets}{$marketId}{orderIndex} =
                $response->{'marketItems'}{'n2:MarketSummary'}{orderIndex}{'content'};
                $events_hash{markets}{$marketId}{marketType} =
                $response->{'marketItems'}{'n2:MarketSummary'}{marketType}{'content'};
                $events_hash{markets}{$marketId}{startTime}  =
                $response->{'marketItems'}{'n2:MarketSummary'}{startTime}{'content'};
                $events_hash{markets}{$marketId}{eventTypeId}        =
                $response->{'marketItems'}{'n2:MarketSummary'}{eventTypeId}{'content'};
                }
        if ( $type eq "ARRA" )
        {
        foreach my $s (@{$response->{'marketItems'}{'n2:MarketSummary'}})
```

```
                {
                $events_hash{markets}{$s->{marketId}{'content'}}{timezone}  =
                $s->{timezone}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{menuLevel} =
                $s->{menuLevel}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{marketName}        =
                $s->{marketName}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{orderIndex}        =
                $s->{orderIndex}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{marketType}        =
                $s->{marketType}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{startTime} =
                $s->{startTime}{'content'};
                $events_hash{markets}{$s->{marketId}{'content'}}{eventTypeId}       =
                $s->{eventTypeId}{'content'};
                }
        }
        if ( $type eq "HASH" )
                {
                my $eventId=$response->{'eventItems'}{'n2:BFEvent'}{'eventId'}{'content'};
                $events_hash{events}{$eventId}{orderIndex}    =
                $response->{'eventItems'}{'n2:BFEvent'}{orderIndex}{'content'};
                $events_hash{events}{$eventId}{eventName}     =
                $response->{'eventItems'}{'n2:BFEvent'}{eventName}{'content'};
                $events_hash{events}{$eventId}{timezone}      =
                $response->{'eventItems'}{'n2:BFEvent'}{timezone}{'content'};
                $events_hash{events}{$eventId}{startTime}     =
                $response->{'eventItems'}{'n2:BFEvent'}{startTime}{'content'};
                $events_hash{events}{$eventId}{menuLevel}     =
                $response->{'eventItems'}{'n2:BFEvent'}{menuLevel}{'content'};
                $events_hash{events}{$eventId}{eventTypeId}  =
                $response->{'eventItems'}{'n2:BFEvent'}{eventTypeId}{'content'};
                }
        if ( $type eq "ARRA" )
        {
        foreach my $s (@{$response->{'eventItems'}{'n2:BFEvent'}})
                        {
                        $events_hash{events}{$s->{eventId}{'content'}}{orderIndex}
                        =$s->{orderIndex}{'content'};
                        $events_hash{events}{$s->{eventId}{'content'}}{eventName}
                        =$s->{eventName}{'content'};
                        $events_hash{events}{$s->{eventId}{'content'}}{timezone}
                        =$s->{timezone}{'content'};
                        $events_hash{events}{$s->{eventId}{'content'}}{startTime}
                        =$s->{startTime}{'content'};
                        $events_hash{events}{$s->{eventId}{'content'}}{menuLevel}
                        =$s->{menuLevel}{'content'};
                        $events_hash{events}{$s->{eventId}{'content'}}{eventTypeId}
                        =$s->{eventTypeId}{'content'};
                        }
        }
        return %events_hash;
        }

sub get_market_prices_compressed
        {
  my ($sessionToken,$marketId)=@_;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getMarketPricesCompressed
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <marketId>'.$marketId.'</marketId>
                        </m:request>
```

```
            </m:getMarketPricesCompressed>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "get_market_prices_compressed failed
        to retrieve valid XML"};

  my %prices_hash;
#  print Dumper(\$result);
          my $response=\%{$result-
        >{'soap:Body'}{'n:getMarketPricesCompressedResponse'}{'n:Result'}};
#  print Dumper($response);
        $prices_hash{'errorCode'}            =
        $response->{'errorCode'}{'content'};
        $prices_hash{'timeStamp'}            =
        $response->{'header'}->{'timestamp'}{'content'};
        $prices_hash{'sessionToken'}         =
        $response->{'header'}->{'sessionToken'}{'content'};
my $runner_price= $result->
        {'soap:Body'}{'n:getMarketPricesCompressedResponse'}{'n:Result'}{'marketPrices'}{'
        content'};
        $runner_price=~ s/\\:/colon/g;
#       print $runner_price,"\n";
        $prices_hash{'noRunners'}=0;
        my @price_split=split(/:/,$runner_price);
        my $size=@price_split;
#       print $size,"\n";
        my @market_attributes=split(/\~/,$price_split[0]);
        $prices_hash{'marketId'}             =$market_attributes[0];
        $prices_hash{'currencyCode'}         =$market_attributes[1];
        $prices_hash{'delay'}                =$market_attributes[3];
        $prices_hash{'marketStatus'}         =$market_attributes[2];
        $prices_hash{'marketInfo'}           =$market_attributes[5];
        $prices_hash{'numberOfWinners'}      =$market_attributes[4];
        $prices_hash{'lastRefresh'}          =$market_attributes[8];
        $prices_hash{'IsBSP'}                =$market_attributes[10];
        #returns a value of Y or N

        for ( my $t =1 ; $t < $size ; $t++)

        {
                my @prices =split(/\|/,$price_split[$t]);
                my @header =split(/\~/,$prices[0]);
                my @back   =split(/\~/,$prices[1]);
                my @lay    =split(/\~/,$prices[2]);
                $prices_hash{'noRunners'}++;
                my $runnerId = $header[0];
                $prices_hash{'prices'}{$runnerId}{'orderIndex'}             =$header[1];
                $prices_hash{'prices'}{$runnerId}{'totalAmountMatched'}     =$header[2];
                $prices_hash{'prices'}{$runnerId}{'lastPriceMatched'}       =$header[3];
                $prices_hash{'prices'}{$runnerId}{'asianHandicap'}          =$header[4];
                $prices_hash{'prices'}{$runnerId}{'reductionFactor'}        =$header[5];
                $prices_hash{'prices'}{$runnerId}{'vacantTrap'}             =$header[6];
                $prices_hash{'prices'}{$runnerId}{'farBSP'}                 =$header[7];
                $prices_hash{'prices'}{$runnerId}{'nearBSP'}                =$header[8];
                $prices_hash{'prices'}{$runnerId}{'actualBSP'}              =$header[9];
                $prices_hash{'prices'}{$runnerId}{'backDepth'}=0;
                my $back_size=@back;
                my $lay_size =@lay;
```

```
if ( $back_size == 0 )
        {
        $prices_hash{'prices'}{$runnerId}{'backDepth'}=0;
        }
if ( $back_size == 4 )
        {
        $prices_hash{'prices'}{$runnerId}{'backDepth'}=1;
        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'}=$back[0];

        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'}=$
        back[1];
        }
if ( $back_size == 8 )
        {
        $prices_hash{'prices'}{$runnerId}{'backDepth'}=2;
        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'}= $back[0];

        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'}=
        $back[1];
        $prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'price'}= $back[4];
        $prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'amountAvailable'}=
        $back[5];
        }
if ( $back_size == 12 )
        {
        $prices_hash{'prices'}{$runnerId}{'backDepth'}=3;
        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'price'}=$back[0];
        $prices_hash{'prices'}{$runnerId}{'back'}{'1'}{'amountAvailable'}
        =$back[1];
        $prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'price'}=$back[4];
        $prices_hash{'prices'}{$runnerId}{'back'}{'2'}{'amountAvailable'}=
        $back[5];
        $prices_hash{'prices'}{$runnerId}{'back'}{'3'}{'price'}=$back[8];
        $prices_hash{'prices'}{$runnerId}{'back'}{'3'}{'amountAvailable'}=
        $back[9];
        }
if ( $lay_size == 0 )
        {
        $prices_hash{'prices'}{$runnerId}{'layDepth'}=0;
        }
if ( $lay_size == 4 )
        {
        $prices_hash{'prices'}{$runnerId}{'layDepth'}=1;
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}=$lay[0];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=
        $lay[1];
        }
if ( $lay_size == 8 )
        {
        $prices_hash{'prices'}{$runnerId}{'layDepth'}=2;
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}=$lay[0];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=
        $lay[1];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'price'}=$lay[4];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'amountAvailable'}=
        $lay[5];
        }
if ( $lay_size == 12 )
        {
        $prices_hash{'prices'}{$runnerId}{'layDepth'}=3;
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'price'}=$lay[0];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'1'}{'amountAvailable'}=
        $lay[1];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'price'}=$lay[4];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'2'}{'amountAvailable'}=
        $lay[5];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'3'}{'price'}=$lay[8];
        $prices_hash{'prices'}{$runnerId}{'lay'}{'3'}{'amountAvailable'}=
        $lay[9];
        }
}
```

```
        my $prices = 'prices';

        return %prices_hash;


}


sub place_bet
        {
        my %bet_hash = ();
        my
        ($sessionToken,$asianLineId,$betType,$marketId,$price_asked,$selectionId,$stake,$b
        etCategoryType,$betPersistenceType,$bspLiability)=@_;
        my $xml='<SOAP-ENV:Envelope xmlns:SOAP-
        ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
        ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:m0="http://www.betfair.com/publicapi/types/">
        <SOAP-ENV:Body>
                <m:placeBets
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <bets>
                                        <m0:PlaceBets>
                                <asianLineId>'.$asianLineId.'</asianLineId>
                                <betType>'.$betType.'</betType>
                                <betCategoryType>'.$betCategoryType.'</betCategoryType>

                <betPersistenceType>'.$betPersistenceType.'</betPersistenceType>
                                <marketId>'.$marketId.'</marketId>
                                <price>'.$price_asked.'</price>
                                <selectionId>'.$selectionId.'</selectionId>
                                <size>'.$stake.'</size>
                                        <bspLiability>'.$bspLiability.'</bspLiability>
                                        </m0:PlaceBets>
                                </bets>
                        </m:request>
                </m:placeBets>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
#print Dumper(\$resp);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "place_bet failed to retrieve valid
        XML"};
my $response=\%{$result->{'soap:Body'}{'n:placeBetsResponse'}{'n:Result'}};
#print Dumper($response);

        $bet_hash{'errorCode'}          =$response->{'errorCode'}{'content'};
        $bet_hash{'timeStamp'}          =$response->{'header'}->{'timestamp'}{'content'};
        $bet_hash{'sessionToken'}       =$response->{'header'}->{'sessionToken'}{'content'};
#my $bet_results;
#$bet_hash{$bet_results} =$response->{'betResults'}{'n2:PlaceBetsResult'};
my $bet_results=$response->{'betResults'}{'n2:PlaceBetsResult'};
```

```
                $bet_hash{'betId'}            =$bet_results->{'betId'}{'content'};
                $bet_hash{'averagePriceMatched'}      =
                $bet_results->{'averagePriceMatched'}{'content'};
                $bet_hash{'resultCode'}       =$bet_results->{'resultCode'}{'content'};
                $bet_hash{'sizeMatched'}      =$bet_results->{'sizeMatched'}{'content'};
                $bet_hash{'success'}          =$bet_results->{'success'}{'content'};
        return %bet_hash;
        }

sub update_bet
        {
        my %update_hash = ();
        my ($sessionToken,$betId,$newPrice,$newSize,$oldPrice,$oldSize, $persistType,
        $oldPersistType)=@_;
        my $xml='<SOAP-ENV:Envelope xmlns:SOAP-
        ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
        ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:m0="http://www.betfair.com/publicapi/types/">
        <SOAP-ENV:Body>
                <m:updateBets
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <bets>
                                        <m0:UpdateBets>
                                                <betId>'.$betId.'</betId>

                <newBetPersistenceType>'.$persistType.'</newBetPersistenceType>
                                                <newPrice>'.$newPrice.'</newPrice>
                                                <newSize>'.$newSize.'</newSize>

                <oldBetPersistenceType>'.$oldPersistType.'</oldBetPersistenceType>
                                                <oldPrice>'.$oldPrice.'</oldPrice>
                                                <oldSize>'.$oldSize.'</oldSize>
                                        </m0:UpdateBets>
                                </bets>
                        </m:request>
                </m:updateBets>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result = XMLin($content);
#print Dumper(\$result);
my $response=\%{$result->{'soap:Body'}{'n:updateBetsResponse'}{'n:Result'}};
#print Dumper(\$response);
        $update_hash{'errorCode'}     =$response->{'errorCode'}{'content'};
        $update_hash{'timeStamp'}     =$response->{'header'}->{'timestamp'}{'content'};
        $update_hash{'sessionToken'}  =$response->{'header'}->{'sessionToken'}{'content'};
my $bet_results=$response->{'betResults'}{'n2:UpdateBetsResult'};
                        $update_hash{'success'}=$bet_results->{'success'}{'content'};
                        $update_hash{'sizeCancelled'}         =
                        $bet_results->{'sizeCancelled'}{'content'};
                        $update_hash{'newPrice'}              =
                        $bet_results->{'newPrice'}{'content'};
                        $update_hash{'oldBetId'}              =
```

```
                        $bet_results->{'betId'}{'content'};
                        $update_hash{'newBetId'}              =
                        $bet_results->{'newBetId'}{'content'};
                        $update_hash{'newSize'}                       =
                        $bet_results->{'newSize'}{'content'};
        return %update_hash;
        }

sub cancel_bet          #use cancel bets by market to close out an entire position
        {
        my %cancel_hash = ();
        my ($sessionToken,$betId)=@_;
        my $xml='<SOAP-ENV:Envelope xmlns:SOAP-
        ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
        ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:m0="http://www.betfair.com/publicapi/types/">
        <SOAP-ENV:Body>
                <m:cancelBets
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <bets>
                                        <m0:CancelBets>
                                                <betId>'.$betId.'</betId>
                                        </m0:CancelBets>
                                </bets>
                        </m:request>
                </m:cancelBets>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result = XMLin($content);
my $response=\%{$result->{'soap:Body'}{'n:cancelBetsResponse'}{'n:Result'}};
#print Dumper(\$response);
        $cancel_hash{'errorCode'}     =$response->{'errorCode'}{'content'};
        $cancel_hash{'timeStamp'}     =$response->{'header'}->{'timestamp'}{'content'};
        $cancel_hash{'sessionToken'}  =$response->{'header'}->{'sessionToken'}{'content'};
        my $bet_results =  $response->{'betResults'}{'n2:CancelBetsResult'};
        $cancel_hash{'resultCode'}    =$bet_results->{'resultCode'}{'content'};
        $cancel_hash{'betId'}         =$bet_results->{'betId'}{'content'};
        $cancel_hash{'sizeMatched'}   =$bet_results->{'sizeMatched'}{'content'};
#for any amounts that got away while making the call
        $cancel_hash{'sizeCancelled'} =$bet_results->{'sizeCancelled'}{'content'};
        return %cancel_hash;
        }


sub get_detailed_market_depth         {
  my ($sessionToken,$marketId,$selectionId)=@_;
        my %depth_hash;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
```

```
                <m:getDetailAvailableMktDepth
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientstamp>0</clientstamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <marketId>'.$marketId.'</marketId>
                                <selectionId>'.$selectionId.'</selectionId>
                        </m:request>
                </m:getDetailAvailableMktDepth>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);

#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "get_detailed_market_depth failed to
        retrieve valid XML"};

my $response=\%{$result-
        >{'soap:Body'}{'n:getDetailAvailableMktDepthResponse'}{'n:Result'}};
#print Dumper($response);
        $depth_hash{'errorCode'}      =$response->{'errorCode'}{'content'};
        $depth_hash{'timeStamp'}      =$response->{'header'}->{'timestamp'}{'content'};
        $depth_hash{'sessionToken'}   =$response->{'header'}->{'sessionToken'}{'content'};

        $depth_hash{'oddsArray'} = $response->{'priceItems'}{'n2:AvailabilityInfo'};

#       return;
#my $array_odds = $response{'priceItems'};
#my $array_odds = $response{'priceItems'}{'n2:AvailabilityInfo'};
#my @array_odds = $response{'priceItems'}{'n2:AvailabilityInfo'};
#my $array_odds = $response{'priceItems'}{'n2:AvailabilityInfo'};

return %depth_hash;


};

sub get_market_traded_volume  {
  my ($sessionToken,$marketId,$selectionId)=@_;
        my %traded_hash;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getMarketTradedVolume
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientstamp>0</clientstamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <marketId>'.$marketId.'</marketId>
                                <selectionId>'.$selectionId.'</selectionId>
                        </m:request>
                </m:getMarketTradedVolume>
        </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;

my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die "get_market_traded_volume failed to
        retrieve valid XML"};

my $response=\%{$result->{'soap:Body'}{'n:getMarketTradedVolumeResponse'}{'n:Result'}};
#print Dumper($response);

        $traded_hash{'errorCode'}     =$response->{'errorCode'}{'content'};
        $traded_hash{'timeStamp'}     =$response->{'header'}->{'timestamp'}{'content'};
        $traded_hash{'sessionToken'}  =$response->{'header'}->{'sessionToken'}{'content'};
        $traded_hash{'volArray'} = $response->{'priceItems'}{'n2:VolumeInfo'};


return %traded_hash;

};

sub get_bet
#example of a subscription API service, bet details can also be retrieved using the Free
        Access API
        {
        my %getbet_hash = ();
  my ($sessionToken, $betId)=@_;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getBet
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>2147463847</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <betId>'.$betId.'</betId>
                        </m:request>
                </m:getBet>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');

$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die};
#print Dumper(\$result);
my $response=\%{$result->{'soap:Body'}{'n:getBetResponse'}{'n:Result'}};
#  print Dumper(\$response);
```

```
        $getbet_hash{'errorCode'}     =$response->{'errorCode'}{'content'};
        $getbet_hash{'timeStamp'}     =$response->{'header'}->{'timestamp'}{'content'};
        $getbet_hash{'sessionToken'}  =$response->{'header'}->{'sessionToken'}{'content'};

        $getbet_hash{profitAndLoss}=$response->{'bet'}{'profitAndLoss'}{'content'};
        $getbet_hash{marketId}=$response->{'bet'}{'marketId'}{'content'};
        $getbet_hash{matchedSize}=$response->{'bet'}{'matchedSize'}{'content'};
        $getbet_hash{matchedDate}=$response->{'bet'}{'matchedDate'}{'content'};
        $getbet_hash{selectionName}=$response->{'bet'}{'selectionName'}{'content'};
        $getbet_hash{selectionId}=$response->{'bet'}{'selectionId'}{'content'};
        $getbet_hash{requestedSize}=$response->{'bet'}{'requestedSize'}{'content'};
        $getbet_hash{avgPrice}=$response->{'bet'}{'avgPrice'}{'content'};
        $getbet_hash{marketName}=$response->{'bet'}{'marketName'}{'content'};
        $getbet_hash{fullMarketName}=$response->{'bet'}{'fullMarketName'}{'content'};
        return %getbet_hash;
        };

sub get_mubets
        {
 my %mubets_hash;
 my ($sessionToken, $marketId, $betId)=@_;   #need to specify @betIds as a third
        parameter in time
unless ($betId == 0) {$betId = "<betId>$betId</betId>"};   #quick fix for single entries
 my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
               <m:getMUBets
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                       <m:request>
                              <header>
                                     <clientStamp>2147463847</clientStamp>
                                     <sessionToken>'.$sessionToken.'</sessionToken>
                              </header>
                              <betStatus>MU</betStatus>
                              <marketId>'.$marketId.'</marketId>

                              <betIds>'.$betId.'</betIds>

                              <orderBy>PLACED_DATE</orderBy>
                              <sortOrder>DESC</sortOrder>
                              <recordCount>10</recordCount>
                              <startRecord>0</startRecord>
                              <matchedSince>0</matchedSince>
                       </m:request>
               </m:getMUBets>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');

$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result = XMLin($content);
#print Dumper(\$result);
my $response=\%{$result->{'soap:Body'}{'n:getMUBetsResponse'}{'n:Result'}};
#print Dumper($response);
        $mubets_hash{'errorCode'}     =$response->{'errorCode'}{'content'};
        $mubets_hash{'timeStamp'}     =$response->{'header'}->{'timestamp'}{'content'};
        $mubets_hash{'sessionToken'}  =$response->{'header'}->{'sessionToken'}{'content'};
        $mubets_hash{'results'}       =$response->{'bets'}{'n2:MUBet'};
        my $matched_results = $mubets_hash{'results'};
```

```
        #force one value to be an array if only one hash returned

        if (ref( $matched_results )  eq "HASH")    {
#if there is only one value returned, force an anonymous array, so all data can be
#accessed in the same way by programs
        my @matched_array;
        push @matched_array, $matched_results;
        $mubets_hash{'results'}       = \@matched_array;
        }


#print "$matched_status, $matched_size, $matched_price, $betType\n";
return %mubets_hash;

        #access individual bet details as follows:
        #$results = $mubets_hash{'results'}
        #foreach my $ref ( @ { $results } ) {

        #$mu_status = $ref->{betStatus}{content};
        #$mu_size = $ref->{size}{content};
        #$mu_price = $ref->{price}{content};
        #$mu_betType = $ref->{betType}{content};
         #$mu_date' = $ref->{matched_date}{content};
        #$mu_selectionId = $ref->{selectionId}{content};
        #mu_marketId = $ref->{marketId}{content};

        #note the following new categories for API6:
        #$mu_persistence = $ref->{betPersistenceType}{content}
        #$mu_betCat = $ref->{betCategoryType}{content}
        #$mu_bsp = $ref->{bspLiability}{content}

}

sub get_market_pandl
        {
  my ($sessionToken, $marketID)=@_;
        my %pandl_hash = ();
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getMarketProfitAndLoss
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:request>
                                <header>
                                        <clientStamp>2147463847</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <includeSettledBets>false</includeSettledBets>
                                <includeBspBets>true</includeBspBets>
                                <marketID>'.$marketID.'</marketID>
                                <netOfCommission>false</netOfCommission>
                        </m:request>
                </m:getMarketProfitAndLoss>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');
$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result = XMLin($content);
#print Dumper(\$result);
```

```
my $response=\%{$result->{'soap:Body'}{'n:getMarketProfitAndLossResponse'}{'n:Result'}};

        $pandl_hash{'errorCode'}      =$response->{'errorCode'}{'content'};
        $pandl_hash{'timeStamp'}      =$response->{'header'}->{'timestamp'}{'content'};
        $pandl_hash{'sessionToken'} =$response->{'header'}->{'sesssionToken'}{'content'};
        $pandl_hash{'results'} =$response->{'annotations'}{'n2:ProfitAndLoss'};
        $pandl_hash{'commissionApplied'}=$response->{'commissionApplied'}{'content'};
        $pandl_hash{'marketStatus'}=$response->{'marketStatus'}{'content'};
        #print Dumper ($pandl_hash{'results'});

        return %pandl_hash;
        }



sub get_account_statement
        {
        my %statement_hash = ();
  my ($sessionToken, $startDate, $endDate)=@_;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getAccountStatement
        xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                        <m:req>
                                <header>
                                        <clientStamp>0</clientStamp>
                                        <sessionToken>'.$sessionToken.'</sessionToken>
                                </header>
                                <startRecord>0</startRecord>
                                <recordCount>1000</recordCount>
                                <startDate>'.$startDate.'</startDate>
                                <endDate>'.$endDate.'</endDate>
                                <itemsIncluded>ALL</itemsIncluded>
                        </m:req>
                </m:getAccountStatement>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');

$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die};
#print Dumper(\$result);
my $response=\%{$result->{'soap:Body'}{'n:getAccountStatementResponse'}{'n:Result'}};
  #print Dumper(\$response);
#print Dumper($response);
        $statement_hash{'errorCode'}  =$response->{'errorCode'}{'content'};
        $statement_hash{'timeStamp'}  =$response->{'header'}->{'timestamp'}{'content'};
        $statement_hash{'sessionToken'}=
        $response->{'header'}->{'sessionToken'}{'content'};
        $statement_hash{'items'}      =$response->{'items'}{'n2:AccountStatementItem'};



        #$statement_hash{'errorCode'} =$response->{'errorCode'}{'content'};
        #$statement_hash{'timeStamp'} =$response->{'header'}->{'timestamp'}{'content'};
        #$statement_hash{'sessionToken'}=
        #$response->{'header'}->{'sessionToken'}{'content'};
```

```
        #$statement_hash{profitAndLoss}=$response->{'bet'}{'profitAndLoss'}{'content'};
        #$statement_hash{marketId}=$response->{'bet'}{'marketId'}{'content'};
        #$statement_hash{matchedSize}=$response->{'bet'}{'matchedSize'}{'content'};
        #$statement_hash{matchedDate}=$response->{'bet'}{'matchedDate'}{'content'};
        #$statement_hash{selectionName}=$response->{'bet'}{'selectionName'}{'content'};
        #$statement_hash{selectionId}=$response->{'bet'}{'selectionId'}{'content'};
        #$statement_hash{requestedSize}=$response->{'bet'}{'requestedSize'}{'content'};
        #$statement_hash{avgPrice}=$response->{'bet'}{'avgPrice'}{'content'};
        #$statement_hash{marketName}=$response->{'bet'}{'marketName'}{'content'};
        #$statement_hash{fullMarketName}=$response->{'bet'}{'fullMarketName'}{'content'};
        return %statement_hash;
        };


sub get_account_funds
        {
         my %funds_hash = ();
  my ($sessionToken)=@_;
#  my $sessionToken = $_;
  my $xml='<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<m:getAccountFunds xmlns:m="http://www.betfair.com/publicapi/v5/BFExchangeService/">
                                <request>
<header>
<clientStamp>0</clientStamp>
<sessionToken>'.$sessionToken.'</sessionToken>
</header>
                                </request>
</m:getAccountFunds>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
my $userAgent = LWP::UserAgent->new();
my $request = HTTP::Request->new(POST =>
        'https://api.betfair.com/exchange/v5/BFExchangeService');

$request->header(SOAPAction =>
        '"https://api.betfair.com/exchange/v5/BFExchangeService"');
$request->content($xml);
$request->content_type("text/xml; charset=utf-8");

my $resp = $userAgent->request($request);
my $content=$resp->content;
#print Dumper(\$content);
my $result;
eval { $result = XMLin($content) };
if ($@) {print Dumper ($content); print "$@\n"; die};
#print Dumper(\$result);
my $response=\%{$result->{'soap:Body'}{'n:getAccountFundsResponse'}{'n:Result'}};
  #print Dumper(\$response);
#print Dumper($response);
        $funds_hash{'errorCode'}            =$response->{'errorCode'}{'content'};
        $funds_hash{'timeStamp'}            =$response->{'header'}-
        >{'timestamp'}{'content'};
        $funds_hash{'sessionToken'}         =$response->{'header'}-
        >{'sessionToken'}{'content'};
        $funds_hash{'availBalance'}         =$response->{'availBalance'}{'content'};
        $funds_hash{'balance'}              =$response->{'balance'}{'content'};


return %funds_hash;
}
```

Appendix 3: Betfair API services

Table A3-1 below lists all Betfair API Services, or calls which can be used for automating betting strategies, with corresponding access levels for each depending upon the user's subscription level. The most commonly used services are described in Chapter 5 and example Perl functions for accessing these are detailed in Appendix 2.

All the example programs described in the book can be run using the services covered by the examples provided and the Free Access API, which is called by specifying a product code of "82" in the **Login** API service, along with the Betfair account username and password for any active account, as described in Chapter 5.

Other API services can be accessed by using the methods outlined in Chapter 5 and using the examples provided as a template.

A listing of services and access levels is also maintained at the Betfair Developers Program (BDP) website accessible from *www.betfair.com*.

Table A3-1: Listing of Betfair API Services

Betfair API Service Name	Personal Full Access		Free Access
	Read-Only	Trans.	Trans.
Login	✓	✓	24 p/m
Logout	✓	✓	✓
KeepAlive	✓	✓	✓
Convert Currency	✓	✓	
Get Active Event Types	✓	✓	✓
Get All Currencies	✓	✓	
Get All Event Types	✓	✓	✓
Get All Markets	✓	✓	✓
Get Bet	✓	✓	
Get Bet History	✓	✓	1 p/m
Get Bet Lite	✓	✓	
Get Bet Matches Lite	✓	✓	
Get Current Bets	✓	✓	60 p/m
Get Current Bets Lite	✓	✓	
Get Detail Available Market Depth	✓	✓	60 p/m
Get Events	✓	✓	✓
Get In-Play Markets	✓	✓	

Table A3-1: Listing of Betfair API Services (continued)

Get Silks	✓	✓	
Get Market	✓	✓	5 p/m
Get Market Info	✓	✓	5 p/m
Get Market Prices	✓	✓	10 p/m
Get Market Prices Compressed	✓	✓	60 p/m
Get Complete Market Prices Compressed	✓	✓	60 p/m
Get Matched and Unmatched Bets	✓	✓	60 p/m
Get Matched and Unmatched Bets Lite	✓	✓	
Get Market Profit And Loss	✓	✓	60 p/m
Get Market Traded Volume	✓	✓	60 p/m
Cancel Bets		✓	60 p/m
Cancel Bets By Market		✓	
Place Bets		✓	60 p/m
Update Bets		✓	60 p/m
Add Payment Card	✓	✓	✓
Create Account			
Delete Payment Card	✓	✓	✓
Deposit From Payment Card	✓	✓	✓
Forgot Password	✓	✓	
Get Account Funds	✓	✓	1 p/m
Get Account Statement	✓	✓	1 p/m
Get Payment Card	✓	✓	✓
Get Subscription Info	✓	✓	✓
Get Refer and Earn	✓	✓	✓
Modify Password	✓	✓	
Modify Profile	✓	✓	✓
Retrieve LIMB Message	✓	✓	
Self Exclude	✓	✓	✓
Submit LIMB Message	✓	✓	
Transfer Funds		✓	✓
Update Payment Card			
View Profile	✓	✓	✓
Withdraw To Payment Card		✓	✓
Service Announcements	✓	✓	✓

p/m = per minute

Appendix 4: Betfair Course Abbreviations

This appendix lists the abbreviations for UK and Irish racecourses used by Betfair, together with the full course name. The abbreviations are represented below as a Perl data hash (but could equally be a data structure within any other programming language), where the value on the left is the lookup key, => is the assignment operator, and the value on the right is the return value.

This data hash is used in Chapter 5, *Example 5-2*, in order to return full course names for the abbreviations found via the Betfair API, for populating coursename descriptions for an example database table, **`races`**.

```
%bf_course_abbrevs = (
Aint    =>      "Aintree",
Ascot   =>      "Ascot",
Ayr     =>      "Ayr",
Bang    =>      "Bangor",
Bath    =>      "Bath",
Bellew  =>      "Bellewstown",
Bev     =>      "Beverley",
Brig    =>      "Brighton",
Carl    =>      "Carlisle",
Cart    =>      "Cartmel",
Catt    =>      "Catterick",
Chelt   =>      "Cheltenham",
Chep    =>      "Chepstow",
Chest   =>      "Chester",
Donc    =>      "Doncaster",
Epsm    =>      "Epsom",
Extr    =>      "Exeter",
Fake    =>      "Fakenham",
Folk    =>      "Folkestone",
Font    =>      "Fontwell",
GLghs   =>      "Great_Leighs",
Good    =>      "Goodwood",
Ham     =>      "Hamilton",
Hayd    =>      "Haydock",
Here    =>      "Hereford",
Hex     =>      "Hexham",
Hunt    =>      "Huntingdon",
Kelso   =>      "Kelso",
Kemp    =>      "Kempton",
Leic    =>      "Leicester",
Ling    =>      "Lingfield",
Ludl    =>      "Ludlow",
MrktR   =>      "Market_Rasen",
Muss    =>      "Musselburgh",
Newb    =>      "Newbury",
Newc    =>      "Newcastle",
Newm    =>      "Newmarket",
Newt    =>      "Newton_Abbot",
Nott    =>      "Nottingham",
Perth   =>      "Perth",
Plump   =>      "Plumpton",
Ponte   =>      "Pontefract",
Redc    =>      "Redcar",
Ripon   =>      "Ripon",
Salis   =>      "Salisbury",
Sand    =>      "Sandown",
Sedge   =>      "Sedgefield",
Sthl    =>      "Southwell",
Strat   =>      "Stratford",
Sligo   =>      "Sligo",
```

```
Taun    =>      "Taunton",
Thirsk  =>      "Thirsk",
Towc    =>      "Towcester",
Uttox   =>      "Uttoxeter",
Warw    =>      "Warwick",
Weth    =>      "Wetherby",
Winc    =>      "Wincanton",
Wind    =>      "Windsor",
Wolv    =>      "Wolverhampton",
Worc    =>      "Worcester",
Yarm    =>      "Yarmouth",
York    =>      "York",
Ballin  =>      "Ballinrobe",
Belle   =>      "Bellewstown",
Clon    =>      "Clonmel",
Cork    =>      "Cork",
DownR   =>      "Down_Royal",
DownP   =>      "Downpatrick",
Dund    =>      "Dundalk",
Fairy   =>      "Fairyhouse",
Gal     =>      "Galway",
GowP    =>      "Gowran_Park",
Killar  =>      "Killarney",
Kilb    =>      "Kilbeggan",
Layt    =>      "Layton",
Leop    =>      "Leopardstown",
Lim     =>      "Limerick",
List    =>      "Listowel",
Naas    =>      "Naas",
Navan   =>      "Navan",
Punch   =>      "Punchestown",
Rosc    =>      "Roscommon",
Silgo   =>      "Sligo",
Curr    =>      "The_Curragh",
Curragh =>      "The_Curragh",
Thurl   =>      "Thurles",
Tipp    =>      "Tipperary",
Tral    =>      "Tralee",
Tram    =>      "Tramore",
Wex     =>      "Wexford"
);
```

Figure A4-1: Perl data hash of Betfair racecourse abbreviations

Appendix 5: Betfair Price Increments

Since the decimal odds requested for any bet, back or lay, require the exact price format and range used by Betfair, automated programs must be capable of specifying the values in this range. The below table shows the price increments for Betfair odds, as discussed in Chapter 9.

Table A5-1: Price Increments for Betfair Odds Markets

Decimal Odds Range	Increment
1.01 → 2	0.01
2 → 3	0.02
3 → 4	0.05
4 → 6	0.1
6 → 10	0.2
10 → 20	0.5
20 → 30	1
30 → 50	2
50 → 100	5
100 → 1000	10

In a Perl data structure, this can be represented as an array comprising of the following values (only the first few values are shown below). Such an array can be used to automatically increment or decrement a price, as shown in *Example 9-2*.

```
my @betfair_odds = (
1.01,
1.02,
1.03,
1.04,
1.05,
1.06,
1.07,
1.08,
1.09,
1.10,
Etc…as per increment table up to and including 1000
)
```

Index To Figures, Tables and Examples

Index to Figures

Index to Tables:

Index to Examples

Index