Betfair Trading with R

A guide to using the BetfaiR package

Betwise Limited

Table Of Contents

Chapter 1 - Introduction

The BetfaiR package for R

Any number of languages can be used for implementing an interface to the Betfair API, but there are plenty of reasons to look at R for creating a programming toolbox for betting strategies which use the Betfair API. First, it is a data analysis environment that can be used by novice and expert programmers alike. The basics of the language are easy to apply, there is copious documentation, example code, and the R environment provides an interactive interface that can be used to type and return results, enabling novices to build programs step by step and experts to explore data and test ideas before creating programs. At the same time, R accommodates the creation of complex scripts and packages as any other full featured programming language.

One of the key aims of the book Automatic Exchange Betting -- was to show exactly how to build code and scripts for the Betfair API from scratch, so we stopped short of creating a high level package that hides the detail of how the functions are implemented from the user.

However, with the principles of implementing the Betfair API already being documented, the logical next step is to abstract the detail from the user and make the code for implementing betting strategies as succinct as possible. This means producing a high level package that can minimize the code which the user has to write and remember, to implement all API functions instead of just a subset, and to provide as many ancilliary functions for the analysis of betting markets and development of quantitative strategies as possible. In short, to create an easy to use, high level programming toolbox for creating strategies with the Betfair API.

Perhaps most interesting for Betfair users, R is a language that is designed specifically for data analysis and has become the lingua franca of statisticians in many fields. This includes quantitative finance, with which exchange betting shares so many parallels. The core language includes powerful statistical techniques, as well as data manipulation and visualisation (ie. graphical) capabilities. Additionally, R is a hugely popular language that has a wide user community contributing new packages for all sorts of purposes at an exponential rate. All these features can be leveraged for the benefit of creating and testing betting strategies using the core API functions.

The Perl library for the Betfair API detailed in Automatic Exchange Betting was used as a starting point, together with the comprehensive Betfair API documentation available at:

http://bdp.betfair.com/index.php?option=com_weblinks&catid=59&Itemid=113

(Download the Sports Reference API Guide near the top of the page.)

Bryan Lewis, (co-author or author of many R packages including iqfeed, esperr, rredis, irlba and fls) coded up all the Betfair API functions in BetfaiR so that they can be used at a high level, as well as adding new functions and plot types that act on the return data.

As a result, the open source package available from Betwise offers an intuitive scripting interface to the Betfair API where all function calls to the API use their corresponding Betfair names, together with arguments where relevant. Implementation details, such as saving and returning session tokens with each subsequent call to the Betfair API, are completely handled for the user. Further, it offers the first scripting interface to Betfair that can be used interactively, in R's usual development and prototyping mode, as well as programmatically. As such, we are hopeful that libraries available in R offer the prospect of adoption by

a wider audience than some of the existing lower level libraries available.

Of course, a codebase to help automate infrastructure provides a great short cut to get strategies to market, but does not guarantee profitability once you arrive.

Also, whilst there are similarities with financial markets, there are key differences in sports betting markets that need to be considered, not least of which is the study of - and access to - fundamental data. At Betwise we provide this for horseracing markets over at , and within this guide we make reference to how you can use R both as an interface to the BetfaiR API and to the Smartform database, to provide a uniform programmatic interface through which to implement betting strategies.

Using Betfair Trading with R

Finally, a few notes on the guide itself.

Through the guide we refer to the R package that provides an interface to the Betfair API as betfaiR and the Betfair exchange itself as Betfair. However, the package itself is installed within the R environment as `betfair` - see the next Chapter for detailed installation instructions.

Whilst you can dip in and out of the guide, we suggest following the Chapter structure (especially if you're new to programming with the Betfair API), since the guide offers a logical walkthrough to programming with the API, from logging in to building strategies.

The code examples can be copied and pasted as is into R, paying attention to any prerequisite steps that are mentioned in the text.

Last but not least, if you have any questions or would like to contribute your own code examples or strategies, please join us at:

`http://answers.betwise.net/`

Chapter 2 - Installing R and BetfaiR

R is described variously as a programming language and a data analysis environment. This is because, unlike some programming languages, R can be run interactively as well as at the command line, or in the background.

The Betfair package is a complete set of functions that can be used in R, designed for programming with the Betfair API. To use the BetfaiR package, you'll need both R and BetfaiR installed.

If you're already have R on your system, you can skip ahead past "Installing R" to the section on Installing the BetfaiR package.

Installing R

Whatever way you choose to use R, you'll need to install it. This section will explain briefly how to download and install R, though a simple `google` for "how to install R" will generally yield useful links within the top 5 hits, including Youtube videos with screenshots if you're unsure on following written steps.

You can generally use any of the instructions you find in the top hits online with confidence, but you'll need to be remember that:

a. you are trying to install R *binaries* (you can also build from the source code, but we're assuming that if you want to build R from source you don't need these instructions), and
b. You need to choose the right version of R corresponding to your operating system.

A brief guide to installing R

1. Go to: `http://www.r-project.org/`

2. Follow the link for "Download R" on the home page

3. You will be asked to click a link to the "CRAN mirror" from which to download R - this should be for a location near you.

4. You'll see something like the following text at the top of the download mirror page you have selected:

   Download and Install R *Precompiled binary distributions of the base system and contributed packages, Windows and Mac users most likely want one of these versions of R:*

   - *Download R for Linux*
   - *Download R for MacOS X*
   - *Download R for Windows*

5. Click the link that corresponds to your OS above, and then click the file within the new page to get the latest version of R for your OS.

6. The relevant R binary will start to download.

7. When the download's done, double click the installer file that you have downloaded and follow the instructions when prompted.

8. When you're done, you should be able to start R by double clicking the R icon in Windows and Mac, or by typing "R" at the OS command line (all systems).

Installing the betfaiR package

First, make sure you have a working install of R, as above. You may also wish to create a directory in your local system called `betwise` to make installation more straightforward - we have done so in the example shown below. Then, download the betfaiR package from:

`http://www.betwise.co.uk/betfair`

You'll see links for two choices:

- Unix/linux/mac package
- Windows package

Select the correct package for your operating system by right clicking the relevant link and saving the file to your local machine. In this case we will save the file to our local `betwise` directory. Now, load the package in R (you only have to do this once) according to the OS specific instructions below.

Installing and loading the package in Windows

First, ensure that you have installed R and downloaded the correct package for Windows from the Betwise website, as above.

Assuming you have downloaded the package to the directory `C:\betwise`, open up R for Windows.

Once R has booted and you are at the R command prompt, type:

```
> install.packages("/betwise/betfair_1.0.0.zip", repos=NULL)
```

Now, load the package for the first time - the packages on which betfaiR depends will be loaded at this point. Type:

```
> library(betfair)
```

And wait for the install to finish. You can now use betfaiR. Next time you use R, you'll be able to type `library(betfair)` without having to wait for the dependent packages to load.

Loading the package in Mac OS X

First, ensure that you have installed R and downloaded the correct package for Mac OS X from the Betwise website, as above.

Assuming you have downloaded the package to the directory `/Users/username/betwise`, open up R for Mac OS X, using the R GUI for Mac OS X.

In the GUI window, go to `Packages & Data`, then select `Package Installer`.

Within the new dialog box called `R Package Installer` select `Local Source Package` from the drop down list at the top of the dialog (the default says `CRAN (binaries)`). Now, click the `Install` button towards the bottom right of the dialog box.

A file selection box will appear, navigate within it to the directory where the betfaiR package is installed

(in our example `/Users/username/betwise/betfair_1.0.0.tar.gz`) and select the package file by highlighting it. Click `Open` in the file selection box and the package will be installed, including all dependencies.

When you are done, go to the R command line and type

```
> library(betfair)
```

And wait for the install to finish. Next time you use R, you'll be able to type`library(betfair)` without having to wait for the dependent packages to load.

Loading the package in Linux

First, ensure that you have installed R and downloaded the correct package for Linux from the Betwise website, as above.

Assuming you have downloaded the package to the directory `/home/username/betwise`, open up R for Linux - usually `R` at the shell prompt.

Once R has booted and you are at the R command prompt, type:

```
> install.packages("/home/username/betwise/betfair_1.0.0.tar.gz", repos=NULL)
```

Now, load the package for the first time. Type:

```
> library(betfair)
```

And wait for the install to finish. Next time you use R, you'll be able to type`library(betfair)` without having to wait for the dependent packages to load.

Getting help with the betfaiR package

Please bear in mind that the betfaiR package is free software and comes with absolutely no warranty.

However, for free help, the Betwise Question and Answer forum is the best place to go to get help with the BetfaiR R package - likewise, if you are an R expert (or "wannabe expert") actively using the package, your input there will be much appreciated.

Register for free on the Betwise site and post your questions and answers to:
`http://answers.betwise.net/`

For general help with the R language, there are plenty of forums (and books, including many useful free guides) available, including the official R mailing lists at `http://www.r-project.org/`

Chapter 3 - Login and market search

In this chapter we'll describe how to log in to a Betfair account and search for markets, since these are prerequisites for most typical uses of BetfaiR. We'll also show how to build and use simple scripts within an interactive BetfaiR session to enhance productivity. Newcomers to the R language may find this Chapter particularly helpful in showing how to use R functions, create R objects and extract data.

We'll assume before we start that you have an active Betfair account as well as a working installation of R and BetfaiR on your system (if not or you are unsure, see "Installing R and BetfaiR" in Chapter 1).

Let's get started

Most BetfaiR sessions start naturally by logging in to a Betfair account via the Betfair API (as most web based Betfair sessions start by logging on to the Betfair website interface). Logging in to the Betfair API is a prerequisite to using any other BetfaiR function.

However, before we can log in, we have to ensure that all the functions from the Betfair package are available within our R working environment by loading the Betfair package, as below:

```
> library(betfair)          #load BetfaiR package
```

Now you have the Betfair package loaded, to see a list of all the functions available within the BetfaiR package, you can try this:

```
> ls("package:betfair")
```

Logging in via the API

To login, we need to use the `login` function and, as when accessing the website, supply the username of the Betfair account, the password for that account, and, additionally, the API code in order to access the account programatically. For all accounts, an API access code of '82' can be used (see Appendix 1 for a discussion of BetfaiR API product codes available).

As with other BetfaiR functions, the `login` function name simply adopts the name of the Betfair API call, with necessary, and/or optional arguments. Thus, go to the R command line and substitute the appropriate arguments below with your own user credentials:

```
> login(username, password, api_access_type)
```

Now you should be logged in to BetfaiR and can use any other function in BetfaiR.

Remember, to create variables in R, just type the variable name and assign a value to it, as follows:

```
> username = "mybfaccountusername"
> password = "mybfaccountpassword"
> api_access_type = 22   #for the free API, different codes exist for paid      #use
```

The mechanics of accessing the API securely are hidden from the user with`login` as in every function, such as the web services transaction (which currently uses SOAP) necessary for a secure connection to the exchange.

Further complexity is hidden from the user in the form of maintaining state for the connection with

Betfair, since one of the return values from every Betfair API function (including `login`) is a unique API sessionToken, along with log and error output.

To interact with the betting exchange programmatically, the unique sessionToken is explicitly saved and passed to any subsequent API call. However, in the spirit of keeping the interface simple to use, each sessionToken in BetfaiR is stored behind the scenes for every function call and implicitly supplied to further API functions – so that users can concentrate on what API calls they want to make rather than the mechanics of how they are made.

Therefore, `login` is as simple for the quantitative or programmatic user of BetfaiR as it is for an interactive website user who supplies their user credentials once (which, like the API sessionToken, are then referenced by a cookie token for subsequent transactions).

Enough on the details of how sessionTokens are handled (they just are, so you don't have to worry about it).

Searching Available Markets for Different Sports

Now we are logged in, let's look at interrogating some sports betting markets. Which markets to look at? Well, which market types or *events* are available on Betfair? Try:

```
> event_types = getAllEventTypes()
```

Here we use the function call for getting all event types in BetfaiR and save it to the variable (or object in R parlance) `event_types` (nb. in R you can use `=` or `<-` for assigning values to variables and objects).

Let's look at a subset of the `event_types` available, as returned by our API call - let's say the top 20 event types in the list, using the inbuilt `head` function:

```
> head(event_types, 20)
```

Horse Racing events account for the most actively traded daily markets on Betfair year round, so that's the event type that we're going to drill into, in order to see what individual betting markets are available. To see what markets exist for this event type, try:

```
> horseracing = getAllMarkets(eventTypeIds=list(int=7))
```

Betfair has a vast array of markets available on all sports (ie. event types), and the BetfaiR function `getAllMarkets` will fetch all of them, depending on which event type(s) are supplied as arguments. In this example, we will supply the argument `eventTypeIds` to the function, so as to return event details for the identifier "7". The eventTypeId of 7, as we have just seen, is an identifier used only for horseracing markets currently available in Betfair.

This call is equivalent to going to the Betfair website interface and clicking on *Horse Racing* under the list of *All Sports*, then selecting all events in the various submenus.

Having used the API to retrieve all horsesracing events and save them to the object `horseracing`, we can now manipulate that list of events programmatically.

The first thing to notice about our `horseracing` object is that there are an awful lot of types of market available for each horseracing event. We will typically be interested in markets that meet specific criteria, for the purpose of market analysis and/or betting, trading and arbitrage. How do we select the right type of market programmatically?

Note that no further BetfaiR API calls are needed at this stage, we have all the information we need for the analysis and selection of suitable markets within the local in-memory object `horseracing`, and we can simply leverage R's data manipulation capabilities to extract markets we are interested in.

Our horseracing object contains 16 columns we can search over to find relevant events - we can see what these are by using the `names` function as follows:

```
> names(horseracing)

 [1] "Market ID"            "Market Name"          "Market Type"
 [4] "Market Status"        "Event Date"           "Menu Path"
 [7] "Event Hierachy"       "Bet Delay"            "Exchange Id"
[10] "ISO3 Country Code"    "Last Refresh"         "Number of Runners"
[13] "Number of Winners"    "Total Amount Matched" "BSP Market"
[16] "Turning In Play"
```

So let's narrow down the list of markets that might be interesting - a common search might restrict events to the current day's horseracing markets only, and perhaps to one country only.

There's lots of ways to approach this search. Below, we convert the current system date and the betfair "Event Date" (normally including hours, minutes and seconds) to year-month-day format and then compare them in order to create a filter that can subsequently be applied to meet the criteria of "show today's markets only".

```
> datefilter = format(Sys.Date(),"%Y%m%d") == format(horseracing$"Event Date","%Y%m%d")
```

The object `datefilter` can be used to filter against `horseracing` events to select only those with the current date (we can also easily adapt this for any other object that requires date comparisons).

Next, let's create a filter that can be applied to `horseracing` events to extract those that pertain to a certain country - in the case below, all events in Great Britain:

```
> countryfilter = horseracing$"ISO3 Country Code" == "GBR"
```

Finally, let's combine the two filters and apply them to our object `horseracing` in order to create an object that will only contain details on all horseracing events occurring today in Great Britain:

```
> uk_horseracing_today = horseracing[datefilter & countryfilter, ]
```

That still makes for a large number of markets - in particular, the object includes exotic markets for each race, since each race generally has multiple markets related to it. For example, a single race will typically have different markets for win only betting; place betting; betting without the favourite; forecast, reverse forecast and tricast betting; markets that are one-on-one matches between selected horses from the race, and so on. What if we want to programmatically extract win only markets (generally the most liquid markets in British horseracing), ignoring all other market types?

If we want to find out more about each market, it is easy to display a vector or two from the `uk_horseracing_today` object which tells us more. Descriptions of market types can be found in the "Menu Path" and "Market Name" vectors. Indeed, we can use these vectors this with the larger `horseracing` object too, if we want to identify all markets 'by eye'.

The value of creating and applying filters comes when we want to automatically select certain event types, either to re-use such statements for rapid, interactive programming in BetfaiR, or within automated scripts that can be run without the user's presence. Each of the filters we are showing can be saved and re-used for this purpose.

So, to spot horseracing win markets only for each race, the logic is less direct than with our date and country examples, but we can still automatically extract them. To do so, we'll use the `grep` function (essentially a text search function) in R to explicitly match the market types that we *do not* want. After we filter out all the exotic markets by grepping for them, the ones we have left are all the win markets. Since there is no regular expression to indicate what win markets are, we have to specify all other types instead. The upside of defining this expression is that you'll also see how to specify any of those other market types if you want them. Note that whilst using `grep` differs from the way we created filters for country and date, `grep` would have been another way to acheive the same result.

With `grep`, instead of searching for an exact match of values which return TRUE or FALSE, as with the equality operator `==`, we can search the text in one vector for a sequence of regular expressions and return their location in the vector as a list of values relating to each row. Let's start with the "Menu Path" vector. Typing:

```
> uk_horseracing_today$"Menu Path"
```

will (typically) list tens to hundreds of daily horseracing markets (corresponding to each row of `uk_horseracing_today`) in the UK with different market descriptions (corresponding to each description in the "Menu Path" vector). The first time you go through these Menu Path listings - whatever the sport - you can put together rules that can be used to extract market descriptions for those events you typically want to interrogate or explicitly don't want in future. First time up, this is always a case of manual inspection. In our example, searching through `uk_horseracing_today` for win only markets, we'll assume that we are already familiar with these markets and know that we *do not* want any events which include the following text in the "Menu Path":

```
> unwanted1_filter = grep("antepost|acca|TBP|match|forecast|stall|without|daily",
uk_horseracing_today$"Menu Path", ignore.case=TRUE)
```

Given that we are looking for win only markets, our search would be much easier if the "Menu Path" or "Market Name" contained the words "Win only" or similar - then we could explicitly search for those. Other markets are more helpfully named, but horseracing win markets just carry the description of the event itself (eg. "1m Hcap") rather than including the generic market type. Therefore, we have to exclude those types we don't want above, so now we know the row numbers for these are in the variable `unwanted1_filter`. This filter is insufficient to capture place markets, however, since, unhelpfully, the text "To Be Placed" is included instead in the "Market Name" vector. So we'll use the same principle to exclude these markets also (this time applying our grep to "Market Name"), and save them to a second filter as follows:

```
> unwanted2_filter = grep("place", uk_horseracing_today$"Market Name",
ignore.case=TRUE, ignore.case=TRUE)
```

The exact return values displayed above will vary for each set of daily markets, of course. Note also the option to `ignore.case` in the `grep` function. We include that here since some of the terms we are searching in the "Menu Path" or "Market Name" may be second or third words (and may not therefore be capitalized). There are plenty of other arguments to get `grep` working exactly the way you want - for some pointers (as with every other function in R), try using `args` with the function as in:

```
> args(grep)
```

or, for comprehensive documentation with examples, simply

```
> help(grep)    #nb. ?grep works too
```

Ok, let's return to our search for win markets.

The grep function returned a list of values corresponding to rows in the vectors "Menu Path" and "Market Name" which contained all "exotic" markets. Now we want to save the subset of `uk_horseracing_today` which excludes all these rows (in other words, to return a new vector of rows that *only* contains all win markets). Using subscripting, we can thus return all rows in the `uk_horseracing_today` object which *exclude* this new vector and save these to a new object:

```
> win_markets = uk_horseracing_today[- c(unwanted1, unwanted2), ]
```

Above we concatenate our unwanted rows `c(unwanted1, unwanted2),` and then subtract these rows `[ -` from our `uk_horseracing_today` object.

At last, we have all win horseracing markets that are off today in the UK, saved in a new object `win_markets`.

Surely there has to be a quicker way of extracting this data in future? There is, and this is the approach we'll take when revisiting this example in future.

Using code snippets for greater productivity

Ok, it's time to take stock. We haven't got into analysis of what's going on in markets themselves, or looked at betting opportunities, but the things we've done are sufficiently valuable that it's likely we'll want to re-use them next time we use BetfaiR for any of these tasks.

We've loaded the BetfaiR library, logged in to a Betfair account using the API, previewed all event types and subsequently searched for all horseracing win only markets occurring today in Great Britain (and in doing so covered the general principles of extracting data for any sports market).

Suppose that playing in daily GBR horseracing win only markets is our betting speciality. We now have the commands needed to find such markets programatically, but typing them every time is labourious and remembering the exact syntax is prone to error. Moreover, we may wish to automate these tasks in future.

We can rapidly reduce the time needed both to log in to Betfair with our account details and find these markets by saving the few commands to a file in our working directory and loading that file on the command line the next time we use BetfaiR.

In this case, we will create 2 files, since loading the BetfaiR library and logging in is something we may do separately in future. Indeed, we can build up a library of code snippets for various interactive tasks. We can also re-use these code snippets to build completely automated scripts which are run in batch mode (and do not therefore require starting the R environment).

So, let's leave R (`q()`) and create a file in the local directory that we can load up with commands to be executed by R. We'll call the first file `login.R`. We can use any plain text editor to create it, typing in the following lines of code which mirror what we have already done on the command line:

```
require(betfair)            #load the BetfaiR library, good for this session
user = "username"           #replace with your actual account username
password = "password"       #replace with your real password
code = 82                   #Free API access code - see Appendix A
login(user, password, code) #log in!
```

Quit and save the file, go back into the R environment and simply type at the command line:

```
> source("login.R");
```

In one step we executed all the code already discussed to load the BetfaiR library and get into our account, without having to bother with any of the details.

Let's do the same thing for finding all current daily win horseracing markets in the UK, by creating a script based on the commands already shown to programmatically find these. Again, create a file in our working directory, called `uk_win_markets.R` with the following orderly aggregation of commands we have already discussed:

```
horseracing = getAllMarkets(eventTypeIds=list(int=7));
datefilter = format(Sys.Date(),"%Y%m%d") == format(horseracing$"Event
Date","%Y%m%d");
countryfilter = horseracing$"ISO3 Country Code" == "GBR";
uk_horseracing_today = horseracing[datefilter & countryfilter, ];
unwanted1_filter = grep("antepost|acca|TBP|match|forecast|stall|without|daily",
uk_horseracing_today$"Menu Path", ignore.case=TRUE);
unwanted2_filter = grep("place", uk_horseracing_today$"Market Name",
ignore.case=TRUE);
win_markets = uk_horseracing_today[- c(unwanted1_filter, unwanted2_filter),];
```

Let's save the file and go back into R to see everything working.

This time around, before we let these code snippets do their magic we'll list and then clear out all the objects in the current working environment which may be left over from our previous session:

```
> ls()              #list names of objects in R working environment
> rm(list = ls())   #remove all objects from current working environment
                    #don't do this if you still want to work with these!
> ls()              #no objects left to work with
```

Then we'll run our login code snippet again, which is a prerequisite for searching all markets, followed by our code snippet to create an object for uk win horseracing markets, for today only.

```
> source("login.R");
> source("uk_horseracing_markets.R");
```

If we list all objects again, we'll see all the objects we've previously created immediately at our disposal:

```
> ls()
 [1] "code"                "countryfilter"       "datefilter"
 [4] "horseracing"         "password"            "uk_horseracing_today"
 [7] "unwanted1_filter"    "unwanted2_filter"    "user"
[10] "win_markets"
```

Now `win_markets` itself contains market metadata for each of the daily horseracing markets we may be interested in interrogating further. We can see the available columns with:

```
> names(win_markets)
```

There are all sort of ways we can use this metadata to apply other automated filters for markets we are interested in, such as quickly scanning for highly liquid markets (ie. where large amounts of money have already been traded or matched) as follows:

```
> win_markets[win_markets$"Total Amount Matched" > 500000, ]
```

Or to look for specific types of race (in the case of our horseracing example), such as all 5 furlong sprints (using the "Market Name" field to match "5f"):

```
> win_markets[grep("5f", win_markets$"Market Name"), ]
```

We'll be working further with our final data frame `win_markets` in the next chapter, as we start to retrieve data from markets that are of interest.

Chapter 4 - Getting market data

In this chapter we'll look at how to retrieve live market data from markets that are of interest. (To see how to find market types and events that are of interest - including automating search criteria - refer to Chapter 3). In particular, we will look at retrieving dynamic price data from a market we have already identified, since market price data is a prerequisite to any meaningful market analysis - as well as to betting and trading activity.

As a starting point, let's use the data frame `win_markets` that we created in Chapter 3. This data frame includes all daily win markets in UK and Irish horseracing.

Recall that we can create the daily`win_markets` object by running the following scripts from within the R environment:

```
> source("login.R");                    #See Chapter 3 for details
> source("uk_horseracing_markets.R");   #See Chapter 3 for details
```

From the `win_markets` object we can interrogate events to find those that are of most interest. The critical data element that we need to extract from the data frame that represents each event of interest is the Betfair market ID, since it enables us to retrieve dynamic market data, including prices. This element is explicitly shown in the first column of the `win_markets` data frame with the name "Market ID", with each row relating to a separate market. R also automatically assigns row names (each market is represented by a row) of the Betfair market ID to each row in the data frame. Thus, we can also say:

```
> row.names(win_markets)
```

in order to list all market IDs available in this object. We can therefore save the set of market IDs to a single variable and use one of R's looping constructs to visit each market automatically and then "do" something with each market ID (like get market prices, bet or trade on each market and so on). To do that, we have to know how to perform these operations on one market ID only. Here we look at the case for retrieving market prices for one market ID.

One market ID may be picked 'by eye' after reviewing the list of events manually or via scripted filters, as shown in various examples in Chapter 3. A Betfair market ID may also be supplied automatically from an external source such as Betwise's Smartform database for horseracing data (as discussed in Chapter N), since Smartform already supplies daily Betfair win market IDs, as per the Betfair API, with full market descriptions.

For this example, we'll use the data in our `win_markets` objects to select the most traded market of the day's racing on 6th June 2011 (at a time of 10.44 am BST). To do so we rank the data with the useful R function `order` that orders vectors from lowest to highest ranked. This is typically useful in the context of the BetfaiR package for the purpose of ranking market prices, market volumes and market times (as for ranking the "Event Date"). Thus, to rank all markets in `win_markets` by time, from lowest to highest:

```
> win_markets[order(win_markets$"Event Date"),]
```

To reverse the order, for example to rank by the total amount matched, from highest to lowest (so that the most traded event is shown first) we use a minus - before the variable that is to be ranked (and then save this ranked data frame to a new object) as below:

```
> markets_by_liquidity = win_markets[order(-win_markets$"Total Amount Matched"),]
```

Thus, to get the market ID of the most traded event from our newly ranked data frame we can say:

```
marketId = markets_by_liquidity[1,1]    #first column is market Id
                                        #first row is highest traded
```

This particular example gives us a market ID of 102920485, saved in `market_id`.

Let's get more detailed market data for this market, using the `getMarket` function in BetfaiR, saving the return values to an object, `market_data`.

```
> market_data = getMarket(marketId)
        #or pass in numberic id, ie. getMarket(102920485)
```

This function returns a list of named values, referencing 27 different aspects of the market. Let's find out what they are:

```
> names(market_data)
 [1] "countryISO3"              "discountAllowed"
 [3] "eventTypeId"              "lastRefresh"
 [5] "marketBaseRate"           "marketDescription"
 [7] "marketDescriptionHasDate" "marketDisplayTime"
 [9] "marketId"                 "marketStatus"
[11] "marketSuspendTime"        "marketTime"
[13] "marketType"               "marketTypeVariant"
[15] "menuPath"                 "eventHierarchy"
[17] "numberOfWinners"          "runners"
[19] "unit"                     "maxUnitValue"
[21] "minUnitValue"             "interval"
[23] "runnersMayBeAdded"        "timezone"
[25] "licenceId"                "couponLinks"
[27] "bspMarket"
```

Note that some of these values overlap with the metadata already returned by `getAllMarkets`, which is the function we used to originally create `win_markets`, however `getMarket` is far more detailed. In particular, we get all the static data pertaining to contenders in the event (called `runners` in Betfair parlance, whatever the sport, from horseracing to tennis), along with dynamic data relating to any changes in the market since it was set up.

Note that we can also call `getMarket` and return just one of the columns in one line (using the dollar notation to specify which column we want), as in:

```
> getMarket(marketId)$runners
```

The above returns 4 columns itself. What if we just wanted the names of the runners currently lined up for this market, along with their Betfair `selectionId`? The function call together with results returned are shown below:

```
> getMarket(marketId)$runners[3:4]
                name selectionId
1      Carlton House     5011661
2            Recital     5058628
3          Pour Moi      5461697
4            Seville     5006736
5         Native Khan    4830850
6           Ocean War    4844446
7             Vadamar    5053314
8  Memphis Tennessee     4933859
9      Masked Marvel     4980623
```

```
10    Treasure Beach     4750407
11        Pisco Sour     4830851
12   Marhaba Malyoon     5009750
13 Castlemorris King     4911081
```

Horseracing fans will recognize above the contenders for the 2011 running of the Epsom Derby. Unsurprisingly, this event was the most traded of the day in question up to that point, with over 1 million GBP matched before 11 on the morning of the race (relatively high liquidity for an event with over 5 hours left before the start).

So much for the contenders. Let's see what's going on with current market prices for these. To do so, we need to turn to one of a few functions that enable us to return market prices and volumes for each contender to several levels of market depth - generally it is the level of depth and additional data that changes with each of the market data retrieval functions available.

Let's start by using our basic price retrieval function, `getMarketPrices`.

There are a few possible arguments relating to accounts and markets, but the defaults are generally fine, so that we are only required to supply a valid `marketId`. Thus, to get the current prices for backing or laying and save those (along with other market data variables) to an R object we can say:

```
> market_prices = getMarketPrices(marketId)
```

`getMarketPrices` lists 12 different objects, as follows:

```
> names(market_prices)
[1] "bspMarket"       "currencyCode"    "delay"           "discountAllowed"
[5] "lastRefresh"     "marketBaseRate"  "marketId"        "marketInfo"
[9] "marketStatus"    "numberOfWinners" "removedRunners"  "runnerPrices"
```

There is lots of additional information in here, some of which replicates the market data elements we have already seen returned by other functions. The key element that we are looking for in terms of price data is "runnerPrices", which contains all back and lay prices, together with volumes available for each price, to three levels of depth for the market in question. Thus, we can also say:

```
> market_prices$runnerPrices
```

to return all data that we want. As before, we can be even more succinct with our code in order to capture market prices using the function call and return the prices in just one line, using the function call as follows:

```
> runner_prices = getMarketPrices(marketId)$runnerPrices
```

So `runner_prices` now contains the data we are after. Let's look at this object in more detail. The data structure is a list of runners, each of which contains another list pertaining to details for that runner. Thus, we can see how many runners are competing in the race with

```
> length(runner_prices)
```

Each runner is a named integer within this list. To get all market details for the first runner in the list, we can say:

```
> runner_prices[[1]]
```

Or to get at the back prices only, we can say (with return values shown after the command):

```
> runner_prices[[1]]$bestPricesToBack
  amountAvailable betType depth price
1           48.02       L     1  3.75
2           61.79       L     2   3.7
3           92.52       L     3  3.65
```

Note that the `betType` of `L` relates to the Lay bet that would have been placed into the market, though from our perspective, this now represents an amount that is *available* to back.

Since the function `getMarketPrices` returns a depth of 3 for available back prices and lay prices, all prices are included within the detail for each runner. The depth of 3 relates to prices that are "at the market" (the first row shown) as well as prices that are once and twice removed from the current market level (rows 2 and 3 respectively). Let's look at how to retrieve the current market data only for the back price, back volume, lay price and lay volume.

To do so manually for the first runner in the list, we can say:

```
> runner_prices[[1]]$bestPricesToBack[1,4]
[1] "3.75"  #the best current price available for the first runner in the list
```

To iterate over all these values for each runner, we can use the foreach package in R, loading the library (as with the BetfaiR package), as follows:

```
> library(foreach)
```

Then:

```
foreach(i=runner_prices) %do% i$bestPricesToBack[1,4]
#returns the current best price available for each of the 15 runners
```

If you prefer the prices not in a list, you can concatenate them together with the 'c' function, and save them to a new vector:

```
> best_back_prices = foreach(i=runner_prices, .combine=c) %do%
i$bestPricesToBack[1,4]
```

A list of the best back prices in the market is useful in its own right for all sorts of reasons, such as looking at the market overround. However, it doesn't allow us to easily see which price is available for which runner. Thus, it would be useful, for example, if we were thinking of placing a bet to easily look up the best price for a runner we may be interested in backing, using its selection Id.

To overcome this, we can label each price easily by selection Id, for example, with :

```
> best_back_prices = foreach(i=runner_prices, .combine=c) %do% {x =
as.numeric(i$bestPricesToBack[1,4]); names(x) = i$selectionId; x}
```

So, above, the 'i' variable in foreach iterates through each of the items in`runner_prices`, returning whatever is returned by the program statements inside the curly brackets `{}`, and combining the results together with the .combine function. Note that we also convert each price to a numeric value in the process, as in `as.numeric`.

Now, if we have the relevant runner ID for the contender we are interested in, we can extract the best price available from our new `best_back_prices` object (where "5011661" is the runner ID we are interested in) thus:

```
> best_back_prices[names(best_back_prices)=="5011661"]
```

Other questions about further levels of prices available can be answered similarly. So, "What is the second best lay price available for each selection ID?":

```
secondBest = foreach(i=runner_prices) %do% {
x = as.numeric(i$bestPricesToLay$price)
# Some runners might not have prices or only one price...
if(length(x)>1) {
x = max(x[-which(x==max(x))])
}
x
}
```

Let's also assign names to the secondBest object…

```
names(secondBest) = foreach(i=runner_prices,.combine=c) %do%
i$selectionId
```

As we have seen, the `getMarketPrices` function returns the current back and lay price, plus two further levels of depth. By current price we mean the best price at which the market is guaranteeing available matches - either to back or to lay. Of course, there are many more prices - each to varying degrees of volume - available in the market as a whole. What if we want to see all of the prices in the market, and the volume for each price, for all runners?

This is where `getMarketPrices` needs a big brother, and finds it in the shape of `getCompleteMarketPrices`, which returns all data for every runner in the market.

Using `getCompleteMarketPrices` is just the same as `getMarketPrices`, thus:

```
> full_market_prices = getCompleteMarketPrices(marketId)
```

Use of the return values from `getCompleteMarketPricesCompressed` is a little different from functions we have seen so far, in that our new object `full_market_prices` is a list of lists, where each runner is represented as a list named by its selectionId. Therefore, the names of our returned object will vary according to each market. Two variables, the marketId and the InPlayDelay for the market, are constant. Thus:

```
> names(full_market_prices)
 [1] "marketID"    "inPlayDelay" "4750407"     "4844446"     "4933859"
 [6] "5011661"     "5009750"     "4830850"     "5058628"     "4830851"
[11] "4911081"     "5053314"     "5461697"     "4980623"     "5006736"
```

Our runner object, represented by its Betfair selection Id, is itself a list of lists. Thus, to pick on one selectionId from the above market at random, we can say:

```
> runner = full_market_prices$"4750407"
> names(runner)
 [1] "selectionId"        "OrderIndex"         "TotalAmountMatched"
 [4] "LastPriceMatched"   "Handicap"           "ReductionFactor"
 [7] "Vacant"             "AsianLineId"        "FarSPPrice"
[10] "NearSPPrice"        "ActualSPPrice"      "prices"
```

This time, having saved our runner object and interrogated it, we get back a generic list of named variables that will apply to each runner in all types of market. But what are we looking at?

Simply printing `runner` at the command line will unravel the list of specific values for each variable, most of which are self-explanatory. The full description of each of the returned variables can also be found in the official Betfair API documentation.

Below, we'll concentrate on some particularly interesting variables that provide data not easily acessible elsewhere.

`LastPriceMatched` gives the last price where there was a matched bet on this runner. `TotalAmountMatched` shows us the total volume matched on this runner in this market up to the time of the call `Orderindex` shows us where this runner was ranked in the market at the time of the call (eg. an orderindex of 1 would be the market favourite) `NearSPPrice` and `FarSPPrice` are Betfair's estimates of the likely range of the Betfair SP (which is calculated once the event has gone "in-play") based on the bets taken so far at SP (as opposed to those matched on the exchange).

Of greatest interest is the`prices` variable, which contains all prices and volumes for each point on the Betfair price ladder where there is an amount available to bet, both on the back and lay side of the market, as well as on the Betfair SP market.

We can interrogate this data for a single runner as follows:

```
> runner$prices
```

There are 5 columns of data relating to this runner's price and volume position within the market, with self explanatory headings. There is much that can be done using R to manipulate and analyse the price ladder, as we'll show in the next chapter.

Furthermore, successive calls to `getCompleteMarketPrices` will produce some very interesting data for graphing and analysis purposes, since we will have a time series of price and volume data for each runner. Once we compile prices over regular time intervals we can apply technical analysis to our prices. We'll look at creating such an object in the next section.

Collecting complete market price time series

So far we've only covered single calls to get a snapshot of the market at any given time. Let's create a script that can grab lots of prices dynamically, in order to create a time series object.

There are a few fundamental coding differences to bear in mind when collecting market price time series as opposed to making a single call at a single point in time.

Let's start with the principle of using looping constructs to collect prices. There are several ways to create looping constructs in the R language. The general principle for using a loop in this context is to explicitly get market prices at some predetermined time interval and build up a data structure containing all those prices. If we are using a script to collect prices that is 'set and forget' then we will want to use a loop that continues to collect prices *until* some condition is met.

For example, we could say 'get market prices' at certain intervals (using the`sleep` function) to repeat for a set number of those intervals. Below we do this by specifying 600 iterations (in the loop) of 1 second intervals (by specifying 1 second `sleep` within the body of the loop. This will give us 600 seconds, or 10 minutes of market data, as follows:

```
> for (1:600) {
    #step 1 = get prices
    #step 2 = save the prices to a data structure
```

```
    #we'll cover the code for getting and saving prices in the next #section
    sleep 1
    #step 3 = specify a time interval before the loop continues
  }
```

In the above case, the "until" condition is simply met at the point that the `for` function counter reaches 600. This might help for collecting random batches of prices; however, a generally more useful condition is to collect prices in relation to the time of the event in question. To do this, we need to know what the current time is, and then compare the current time to the time of the event.

Recall that we used the `Sys.Date` function to format and record the current date for comparison purposes in Chapter 3. We can use its close relation, `Sys.time()`, to do the same thing for the current system time as well as the time of the event.

To get the current system time in hours, minutes and seconds we can say:

```
> format(Sys.time(),"%H%M%S"))
```

If we want to use this time to compare against any other time within the same 24 hour period as a numeric value, we can convert the value with the `as.numeric` function, thus:

```
> current_time = as.numeric(format(Sys.time(),"%H%M%S"))
```

We'll also need the time of the event as a value to compare against. Let's select the first event within our `win_markets` object, as follows:

```
> first_race = win_markets$"Event Date"[1]
```

Recall that the "Event Date" in Betfair holds the date and time of the event. If the event we are comparing is within the same 24 hour period we can leave out the date part and format the event time as with the system time above, as follows:

```
> time_of_first_race = as.numeric(format(first_race,"%H%M%S"))
```

Now we can adapt our specified number of iterations with a while condition that compares the event time with the current time, as follows:

```
> while (time_of_first_race > current_time) {
    #step 1 = get prices
    #step 2 = save the prices to a data structure
        #we'll cover the code for getting and saving prices in the next #section
    sleep 1  #interval before loop continues
        #step 4 = update the current time below
    current_time = as.numeric(format(Sys.time(),"%H%M%S"))
    }
```

So much for the principles of using loops to collect market prices. In the next section we'll look at creating new functions and a script that can build the time series structure.

Building a data structure containing market data time series

The script in this section uses a couple of functions for efficiency, which we'll discuss before moving onto the script itself. Creating functions in R is generally an efficient way to code when we want to avoid repetition and enable re-usability. In this case, `getPrices` and `update` will help.

`getPrices` is needed since the script will repeatedly call the `getCompleteMarketPrices` function and pull out the values we want from the resulting data structure. `getPrices` will get the basic values for us from this data structure - ie. the current back and lay price. Note also that we can get any value from `getCompleteMarketPrices` by adapting this function.

`update` is needed since we will repeatedly use the same code to update the data structure with the prices from the last timestamp, adding these prices to the records for each of the runners in the event.

If you're new to R, you shouldn't worry too much about how the functions shown below work - you can simply use them in the script .

The getPrices function

```
# getPrices function to pull out prices after fetching market data
# Pick out the bp1, lp1 prices from the price ladder
# Can easily adjust this to pick out other prices, like bp2, bp3, etc.

getPrices = function(P)
{
    f = as.numeric(P[,1])
    back = c(P[,2])
    lay = c(P[,3])
    idx = back == 0 & lay == 0
    f = f[!idx]
    back = back[!idx]
    lay  = lay[!idx]
    lidx = which(lay>0,arr.ind=TRUE)
    #handle market runners without a lay price
    if(length(lidx)<1) return(list(back=NA,lay=NA))
    layPriceIndex = which(lay>0,arr.ind=T)[[1]]
    backPriceIndex = layPriceIndex - which(back[layPriceIndex:1]<1)[[1]]
    return(list(back=f[backPriceIndex], lay=f[layPriceIndex]))
}
```

The update function

```
# Update the time series 'data' with columns for each runner's at the market
# price and volume

update = function(data, prices, runners)
{
t = getLastTimestamp(TRUE)
v = foreach(r = runners$selectionId) %do%
{
z = prices[[r]]
if(!is.null(z)) {
  v=unlist(c(z$LastPriceMatched,getPrices(z$prices),z$TotalAmountMatched))
}
else {
  v = rep(NA,4)
}
names(v) = paste( rep(runners[runners$selectionId==r,"name"],4),
                  c("LastPriceMatched", "BP1", "LP1", "Volume"))
v
}
v = xts(rbind(unlist(v)),order.by=t)
```

```
if(is.null(data)) return(v)
else return(rbind(data, v))
}
```

Complete script to collect prices

The below script runs as a complete example, including getting the next market ID of the day at the start of the script. This ID (`nm` below) can of course be replaced by any preferred market ID from any sport.

```
# Get the next market id for horse racing (13):

x = getActiveEventTypes()
nm = x[x$id==13,]$nextMarketId
gm = getMarket(nm)
cat("Next market menu path", gm$menuPath," ID=",gm$marketId, "\n")

# The 'runners' data frame can map runner names to selection IDs

runners = getMarket(nm)$runners

# We build up the time series of runner prices and volumes in an xts
# data frame called 'data' which we initialize to NULL.

data = NULL
p = getCompleteMarketPricesCompressed(nm)
cat("This is an example, you can press <CTRL>+C to break out of this loop
early and look at the data in 'data'\n\n")
j = 0

# We loop for 100 seconds of data, or until the race begins, whichever
# comes first:

while(p$inPlayDelay==0 && j < 100){
data = update(data, p, runners)
cat("Number of data points: ",nrow(data)," market time: ",getLastTimestamp(),"\n")
j = j + 1
Sys.sleep(1)
p = getCompleteMarketPricesCompressed(nm)
}
```

Note that the loop that we discussed earlier for collecting prices up until a certain point using`while` is adapted in our script above to continue collecting prices "whilst" the `inPlayDelay` is a value of zero. This effectively does the same thing as collecting prices up to the scheduled time of an event, in that once an event start the `inPlayDelay` is a value greater than zero. Thus, at this point the script will stop.

Conversely, we could use this test in order to collect prices in running (ie. during an event), by first testing for `inPlayDelay>0` and then collecting prices `while(p$inPlayDelay>0)`.

To run this example script interactively, save all the R code above to a file (including the two functions `getPrices` and `update`) and call it say "collect_prices.R".

We can run this code from within the R environment (as with example scripts from previous chapters) by typing:

```
> source("collect_prices.R")
Number of data points:  1  market time:          2011-08-07T12:12:57.245Z
```

```
Number of data points:  2  market time:          2011-08-07T12:12:56.494Z
Number of data points:  3  market time:          2011-08-07T12:12:59.535Z
Number of data points:  4  market time:          2011-08-07T12:13:00.659Z
Number of data points:  5  market time:          2011-08-07T12:13:01.850Z
```

The output from running this script at the command line is also shown above. The output itself is described in the script with the `cat` function. `cat` is used here to print out basic details about each iteration of the loop that is used to add more data points to the data frame at each timestamp.

Once we have completed the loop - either programmatically, as in the case of a market going "inplay" in our scripted example, or manually broken out of it - we can extract time series data for each runner from the resulting data frame.

By default we've called this data frame `data` in our script, but of course it can be any name according to the markets that you are collecting.

Going back to the R prompt, to interrogate the specific data fetched in this script:

```
# This shows us the names of the columns of data we just collected:
names(data)
```

Using this script we will return the following values for each runner (where BP1 stands for the 'at the market' Back Price and LP1 stands for the 'at the market' Lay Price):

```
"Runner name LastPriceMatched"              "Runner name BP1"
"Runner name LP1"                           "Runner name Volume"
```

In our example script, all the columns in the data frame are named after the runner along with the data type being collected, as detailed above.

Thus, we can manipulate data as follows:

```
> runner_name_back_price = data[, "runner name BP1"]
# to rename the back price column for a specific runner as a single vector
# to use for plotting.

# If we just want to look at BP1 for example, we can do this:

> bp1 = data[, grep("BP1",names(data))]
> names(bp1)
```

In the next chapter we'll look at functions in BetfaiR for analysing the market data that we've collected using the above techniques.

Chapter 5 - Market Price Analysis and Visualisation.

This chapter explains how to derive and visualise market data. By now, we assume you're familiar with the function `GetCompleteMarketPricesCompressed`, and are also comfortable with building and manipulating a data frame of market prices by iterating over this function, as explained in the previous chapter.

Visualising the price ladder with plotPrice

For any runner in any market, there will be a series of possible bet amounts available at different prices (according to Betfair price increments - see Appendix 2 for a listing of the prices available).

Visualising the price ladder is a highly useful feature. It's hard to comprehend what an array of prices and different amounts actually mean without visualising them. As such, a visualisation of the price ladder (with different sides of the graphic for back and for lay, for every price where there is a bet amount available) is often provided in expensive third party software built upon the Betfair API. With the BetfaiR R package you get this thrown in, by leveraging the plotting capabilities of R through the BetfaiR function `plotPrice`.

`plotPrice` shows you the price ladder by runner, with two sides of the resulting plot for back and lay prices, with every amount (volume) available shown at every back price and lay price. You can decide upon the "depth" of prices to be shown as an argument to the plot function. For example, a depth of 3 will show all prices and available volumes according to the closest 3 back and lay prices, as per the default Betfair interface. Further, the volume available to bet at each price is shown as a proportional bar, scaled automatically according to all amounts displayed within the plot. This makes it easy to `see` the weight of money, and possibly confidence, behind each runner in a given market, providing a useful visual aid to predicting price movement and market confidence.

A summary of the current back price, the current lay price, the total amount matched so far on the runner in question, as well as the last price matched (as opposed to available) is also displayed on the plot.

To run the `plotPrice` function and create a price ladder visualisation for a runner we're interested in, we'll first need to grab a snapshot of complete runner prices for the market we are interested in visualising, using the `getCompleteMarketPricesCompressed` function. Further, since the price ladder is shown for only one runner at a time, we'll have to specify prices for the runner we are interested in and save this to a new object. We can use an already fetched set of prices and supply the runner ID, or make a call to the Betfair API whilst specifying only the runner we are interested in, for the purposes of running the plot. We'll show the latter method here.

First let's assume you've logged in and have found a relevant Betfair market ID. Then, recall we can run `getCompleteMarketPricesCompressed` with a runner ID attached as follows:

```
> runner_to_plot = getCompleteMarketPricesCompressed(103438705)$"4987995"
# select an active market and a runner's "price ladder" within the market
```

Now we can run `plotPrice` to visualise the price ladder for this runner, which expects two arguments: i) The runner object as above ii) The level of depth to display prices and volumes

Thus,

```
> plotPrice(runner_to_plot, width=7)
```

will give us the graphic we need.

This provides a snapshot of the market as it stands since the last API call for one runner. If we are capturing prices on an active event, we will see the volume (as represented by "Total Amount Matched" in the plot above) increase exponentially for all runners the closer that the event nears the scheduled start or offtime ("the off" as it's known in racing).

But we can do much more with market data, and show how BetfaiR can leverage more cool R packages in the process.

Technical analysis plots with quantmod

Assume that we have run repeated calls to `getCompleteMarketPricesCompressed`, building an array of prices with timestamps, as shown in the Chapter 4. Let's say we have stored back prices for all runners in a market in the variable shown as the default in the example script, `data`.

Now we can interrogate this dataset for the purposes of technical analysis, either whilst the market is still active or as an historic data set.

First, let's load up Jeff Ryan's quantmod package.

```
> require(quantmod)                                # Popular quant package
```

Next we'll pick a runner from `data` and use one of quantmod's nice financial markets plots to chart the prices over time.

```
> Pour_Moi = data[,"Pour Moi"]                     # Pick a runner
> chartSeries(Pour_Moi, ,major.ticks='seconds')
# Classic price plot
```

With the plotting device open, we can calculate and provide appropriate extra detail for this plot on the fly, so to speak, as we'll see in the next few sections.

Calculating the market overround

The implied probabilities of the back prices in a betting market (at least in person to person, electronic markets, such as the Betfair exchange) will tend towards 1. If the sum of implied probabilities in any given market dips below 1, there is an immediate opportunity to make certain money by backing every runner to varying amounts.

Thus, a natural tension between back and lay prices exists for the whole market around 1.0. The percentage by which the market tips above 1.0, usually expressed as 100%, is classically known as the overround in betting markets, in homage to the traditional market makers (ie. bookmakers) who create their own markets and thus their own overround (aka profit margin).

Clearly, if we are thinking of backing a runner, on the whole our odds will be more favourable the closer the market overround stands to 1. If we calculate this, we can also spot those rare "underround" opportunities.

Manually, the overround is impossible to calculate and execute for all runners within the short timeframes that such opportunities become available. Programatically, it's trivial, as follows:

```
# Compute overround
overround = as.xts(rowSums(1/bp1) - 1, order.by=index(bp1))
```

Of course, the one-liner above does more than calculate the overround, it does so for each data point corresponding to the back prices used in our existing plot, as well as representing it as a time series so that we can easily add it as a new indicator.

We'll show how to use this indicator in the next section.

Sophisticated technical analysis plots

Let's not stop there – we can use the technical analysis types already integrated with quantmod to add umpteen number of technical analysis trading indicators. In the example below, we do that, also specifying Bollinger bands and the Relative Strength Indicator:

```
addTA(overround, col=2)                    # Add overround indicator
addBBands()                                # Add Bollinger bands
addRSI()                                   # Add RSI
```

So let's see a quick snapshot of our final technical analysis plot, showing Bollinger bands around the price action, and bottom panels which include the new overround indicator and RSI.

So much for one runner, what about the market as a whole? How are the runner prices behaving with regard to each other?

Another plotting function included in the BetfaiR package is our friend. This function, iprobs, plots the implied probabilities over time for all the runners in the race, taking a dataset of back prices as the input.

The implied probability plot

Once you are into using R with datasets of Betfair prices (such as our example`data` above), there are all sorts of useful plots that you can create to represent that data in interesting one. Here's how to produce a plot we like that illustrates this principle that was put together by Bryan for his talk at RFinance in 2011 (you can see slides from the talk here: http://www.rinfinance.com/agenda/2011/BryanLewis.pdf).

Note that the following script relies on using a suitable dataframe as shown in`data`, generated in the previous Chapter.

```
# Here we make a probability plot:
# First, collect the bp1 prices together:

prices = data[,grep("BP1",names(data))] #create "data" first, of course!
# Probability plot (needs the RColorBrewer package for nice colors)...
require(xts)
require(RColorBrewer)
pr1 = as.matrix(1/prices)
n = length(pr1[,1])
xx = c(1:n,n:1)
z = rep(0,n)
p1=par(xaxs='i',mar=c(4,4,1,2))
plot(xx,xx,type="n",ylim=c(0,1.1),xlab="Time",ylab="Market probabilities",xaxt="n",
    bty='n')
par(p1)
s = z
#clrs = rainbow(s=0.8,n=ncol(prices),start=0,end=4/6)
clrs = brewer.pal(ncol(prices),"BuGn")
for(j in 0:(ncol(pr1)-1))
```

```
{
    k = j+1
    yy = c(s,rev(s + pr1[,k]))
    polygon(xx,yy,col=clrs[k],border=NA)
    s = s + pr1[,k]
}

p1=par(xpd=NA)
legend('top',col=clrs, legend=colnames(prices),ncol=ncol(prices),fill=clrs,cex=0.9,
    bty="n")
par(p1)

idx = index(to.minutes(prices[,1]))
idx = align.time(idx) - 60
idx = format(idx, "%H:%M:%S")
l = nrow(prices)
axis(side=1,line=0,labels=idx,at=seq(1,l,by=l/length(idx)),cex=0.55)
lines(x=c(0,nrow(prices)),y=c(1,1), col='white')
lines(x=c(0,nrow(prices)),y=c(0,0))
title ("Probabilities to Win")
```

What are we looking at? The fatter the band, the greater the share of the market taken by any individual runner, and the lower the price. As implied probability bands shift from fat to thin and vice versa, we can easily spot which runner prices are drifting and which are contracting. Useful indeed if you are into trading with Betfair.

Chapter 6 - Betting and Trading

This chapter shows how to use the BetfaiR package for betting and trading via the BetfaiR API. Here we concentrate on the functions that are used in both betting and trading strategies rather than a discussion of such strategies per se.

In Chapter 7 we'll further illustrate the mechanics demonstrated in this chapter to show some example applications of how these functions can be applied to real world betting and trading strategies.

Betting and Trading - what's the difference?

To trade or to bet involves the same starting point since both a bet and a trade start with a transaction either to back or to lay a contender in an event at a certain price.

A bet typically consists of the initial (back or lay) transaction only and will be left to stand until the outcome of the event is known. This creates an open position (ie. liability) upon a certain contender until the outcome of the event is known, when the position will be converted to a profit or loss.

A trade will typically include a second transaction, as well as the one above which is in the opposite direction of the first (either back or lay) in order to close the position and realise a profit or loss on the trade (ie. each set of two transactions) before an event is started or concluded (in the case of 'in-play' markets). Ideally, the trade will capitalize upon a favourable movement in the price of a contender which means that a profit is made before the outcome of the event is known. In this sense, a trade is a bet upon favourable movements in price in an event, whereas a bet is upon a favourable outcome of the event itself. Several trades can thus be placed upon the same event and upon the same contender in an event, with opportunities to trade both before the event starts and whilst it is 'in-play'. Several bets can also be placed on the same contender in the same event, but typically these will always be in the same direction (ie. to back or to lay, depending on whether the bettor is expecting a win or loss result, respectively).

From the perspective of betting or trading using the BetfaiR package, the mechanics of executing a bet or a trade starts with a single back or lay transaction - ie. a bet.

We'll look next at how to place a bet, either to back or to lay. This can be the starting point of a bet or a trade, or the closing transaction of a trade. Either way, the BetfaiR function used for the transaction is the same.

Placing a bet

Placing a bet using BetfaiR is a simple process, though requires that you know exactly what you want to bet on!

First, we'll assume you have a market that you want to bet in and a runner in that market that you want to bet on.

Next, we'll assume that you know how much you want to bet (ie. stake), whether you want to back or lay the runner in question (ie. bet to win or lose), and, last but not least, what price you want to strike the bet at (it could be an exchange price or, if you are not bothered about a specific price, taking the Betfair SP).

`placeBets` is the function that places your bet for you, provided that you can specify all the above parameters. There are additional parameters that should be specified depending on the market type you may be playing in. In any case the values for these must exist alongside the common ones detailed above.

Thus, a list of parameters might look like this:

```
> asianLineId =         0
#binary value, generally 0 for none Asian handicaps
> betType =             "B"
#"B" to back or "L" to lay
> betCategoryType =         "E"
#"E" for exchange bet (the usual option) or otherwise.  Options are:
#"E", "M" (market on close), or "L" (limit on close)
> betPersistenceType =      "NONE"
#what to do if a bet is unmatched when the event turns in play, options are:
#"NONE", "SP" (convert to starting price), or "IP" (persist in play)
> bspLiability =        0
#default is always 0
> marketId =            103666039
#select a valid market ID
> price =           11.0
#select a price in decimal format for a valid price increment
> selectionId =         3294401
#select a valid market ID
> size =            2.00
#select a stake in decimal format for a valid possible stake
```

Having set our list of parameters, we can now place a bet by listing these out within the`placeBets` function, as follows (note that for this example the variable names are the same as the arguments themselves):

```
> placeBets(list(asianLineId=asianLineId, betType=betType,
betCategoryType=betCategoryType, betPersistenceType=betPersistenceType,
bspLiability=bspLiability, marketId=marketId, price=price,
selectionId=selectionId,size=size))
```

All being well, the function will return the completed bet details as follows:

```
  averagePriceMatched       betId resultCode sizeMatched success
1                21.0 16189019428         OK         2.0    true
```

These bet details should generally be saved to another object, which is necessary in case the bet is not fully matched and we want to subsequently extract the betId to cancel or update the bet at a different price. Indeed, the most important return value for a whole series of operations on bets - from enquiring as to matched, unmatched and partially matched bets to updating and cancelling bets - is the `betId`.

In the meantime, let's return to our `placeBets` example. Whilst the function works fine above, remembering all the parameters and specifying them for more than one bet makes `placeBets` become somewhat unmanageable.

As a result, `newBet` exists as a convenience object that contains all the fields required by placebet. The usage is:

```
> placeBets(newBet(...))
```

So, more conveniently, we can save the individual bet details to an object created by`newBet` (in this case `x`) and then supply them to `placeBets` thereafter as follows:

```
> x = newBet(asianLineId=asianLineId, betType=betType,
betCategoryType=betCategoryType, betPersistenceType=betPersistenceType,
```

```
bspLiability=bspLiability, marketId=marketId, price=price, selectionId=selectionId,
size=size)
```

So we now use `newBet` to build up bet details - and `placeBets` to place them, passing the object created by `newBet` to the latter function:

```
> placeBets(x)
```

Updating a bet

By updating a bet, we mean modifying the parameters of any bet that has not yet been matched. A bet that has not been matched can be modified in terms of either its stake (increasing or decreasing the amount to bet at the price originally specified) or, more commonly, its price - typically in order to ensure that the bet is matched in the exchange market for the original amount.

Further options for updating a bet go beyond price and amount, and relate to the concept of `bet persistence`, such that if a bet has been placed (and is unmatched) on the exchange, it can be modified to persist after the pre-event exchange market has closed, either to be switched to a bet taking the Betfair SP, or to persist during the in-play market, if one is available.

A bet that has already been fully matched is of course no longer modifiable, since there is a counterparty who has taken up the other side of the transaction (in this case, a counter position can be adopted but this will mean making another bet rather than modifying the original one).

The key functions for updating bets using betfaiR are `updateBet` and `updateBets`.

By way of example, let's say that in the St Leger (http://en.wikipedia.org/wiki/St._Leger_Stakes), we like the chances of Census. The current market price on the Betfair exchange is 7.0. We want to make an extremely hopeful exchange bet of 2 gbp at a price of 1000.0 (extremely hopeful since such a bet is highly unlikely to be matched before the event starts), using the following parameters supplied to newBet:

```
> size = 2.00
> marketId = 103709265
> selectionId = 4977051
> betCategoryType = "E"
> betPersistenceType = "NONE"
> bspLiability =0
> price = 1000.0
> #x = newBet(supply parameters above...)
> placeBets(x)
```

Sure enough, a call to placeBets returns the following:

```
> placeBets(x)
averagePriceMatched       betId resultCode sizeMatched success
1               0.0 16261961768         OK         0.0    true
```

So far so good. Now we have an unmatched bet with a `betId` of 16261961768 at 1000.0 on Census.

However, near the event starting the bet still has not been matched, so we decide we will change the status of the bet to Betfair SP if unmatched. In this case, we will no longer be accepting an exchange bet but persisting the bet to be matched at Betfair SP.

To specify this, we need to supply the `betId` to the function `updateBet`. Let's assume that the betId above "16261961768" exists in the variable `betId`, we can update the persistence type as follows:

```
> updateBet(betId, newBetPersistenceType = "BSP", newPrice = NULL, newSize = NULL)
```

Likewise, updating other parameters of the bet (ie. price or size) will result in NULLS for the values that we are not changing. Note that we can only change one aspect of the bet at a time, as per the Betfair documentation.

Cancelling a bet

Alternatively, we can cancel a bet which has not been matched, using `cancelBets`. The form of this function is simple. We need to supply the betId of the bet that we wish to cancel as an argument to the function. Thus, if we have a `betId` 16276631963 for 2 gbp we will cancel as follows:

```
> cancelBets(16276631963)
        betId          resultCode sizeCancelled sizeMatched success
1 16276631963 REMAINING_CANCELLED           2.0         0.0    true
```

Conclusion

We now have all the atomic functions we need to start betting and trading. In the next chapter, we'll look at combining the use of R as an interface to Betfair with fundamental analysis in order to construct fully automated betting strategies.

Chapter 7 - Applying BetfaiR to example betting strategies

This chapter extends the concept of using the BetfaiR package beyond the functions in the package to suggest ways in which complete betting strategies can be built and executed within the R environment.

The examples in this Chapter will only scratch the surface in terms of what is possible, though should provide an insight into ways in which the package can be applied in a wider context.

Whilst a comprehensive discussion of betting strategies is beyond the scope of this guide, such a discussion, along with many detailed examples, is provided in Automatic Exchange Betting (). Likewise, we can only scratch the surface in terms of the fundamental analysis which usually forms the foundation of a betting strategy, but we do explore some examples for horseracing in this Chapter that can be developed using the Smartform database for R, available at .

Betting strategies

A betting strategy typically involves trying to determine the outcome of a certain type of event using a certain, repeatable method.

Using horseracing as an example, predicting an event outcome on the Betfair exchange could be predicting whether a horse will win (ie. should be backed) or lose (ie. should be laid) in a race, whether it will be placed (or not) in a race. It could also be predicting a subset of the event, depending on what other derivative markets exist around the event, such as whether horse A will beat horse B, irrespective of whether or not horse A or horse B wins (or places) in the event itself.

Much discussion around betting strategies concerns whether or not all aspects of the strategy should be executed programmatically, or different programs should be created for each task, some of which may then be run manually.

Whatever the bettor's preference, in starting to look more closely at programs for predicting event outcomes, it is clear we need to use more than market data. Whilst market data can be useful for predicting price and volume activity, it will not assist with assessing gauging the ability of the contenders in an event. Gauging the ability of contenders and how likely the contenders are to show that ability in the peculiar circumstances of the event in question is key to gauging their fundamental chances of success, as opposed to the market's current interpretation of those chances.

Indeed, many parallels exist with the financial markets in this regard, in that the study of price and markets alone can only tell us so much about the value of the financial instrument being traded. To determine the value of the instrument beyond what the current market is willing to pay also requires some data relating to the intrinsic value of the instrument, compared to the trading situation and economy in which the company finds itself. In the case of company shares (ie. equities), data relating to a company's value can be found in its historic and recent trading figures, it's balance sheet - including debt, assets, profit and loss - its operations, its executive management, the success or otherwise of its products, the industry sectors and geographies it operates in and so on.

In the case of horses, it's a similar story, except we are not looking to the historic performance of the company but the historic performance of the horse in order to assess how it might perform in its next event.

Fundamental data in sports betting

In financial markets, copious data relating to the operations of listed companies is used and generally available for quantitative analysts to crunch, in order to come up with investment (ie. betting) strategies. This is generally referred to as fundamental data, and investment strategies which rely on this approach as fundamental investment strategies.

Essentially the purpose of such crunching is to create betting strategies, indicating when it might be a propitious moment to buy or to sell. Such analysis can therefore lead to long term or short term investment recommendations.

This process is analogous to the idea of betting and trading in sports betting markets. Although there is generally an entry and an exit in financial investment (eg. unless buying a derivative and holding it to expiration, such as a call option, which is a straight bet), the principal of adopting either a betting or trading strategy, betting on an outcome or trading in and out of a temporary movement, is the same.

In sports betting our fundamental data are the data relating to the historic performances and the current attributes of all contenders in an event.

In horseracing, this means all the historic performances of horses, including the conditions under which they were achieved, as well as all the data relating to upcoming races and the conditions in which they will be run. At Betwise we license Smartform () as a programming friendly database for the purpose of this type of analysis.

Horseracing analysis using Smartform with R

Since Smartform is loaded in MySQL, it works well with R. To connect to Smartform from R we can use the RMySQL package and establish a connection, represented by `SF` below:

```
> library(RMySQL)  # load library
> SF <- dbConnect(MySQL(), user="smartform", dbname="smartform")
```

There are plenty of ways to query the database now we have established a connection from within R. To maintain a familiar syntax of SQL queries, we can simply save MySQL statements as R objects and then pass them to our connection. So queries to the database will simply take the form of creating an object for each query, using the paste function and including the query statement within it, thus:

```
> sql1 = paste("select distinct(trainer_id) AS 'Trainer ID', trainer_name AS
'Trainer' from daily_runners join daily_races using (race_id) where
meeting_date=CURDATE()", sep="")
```

Now we have statement represented by`sql1` we can pass it to the MySQL connection using the function `dbGetQuery`:

```
> trainer_ids=dbGetQuery(SF, sql1)
> ids=paste(trainer_ids[,1], sep="", collapse=",")
# create comma separated list of all today's trainers
```

Above we have selected the unique trainers for all upcoming races on the current day.

This is a preliminary step to analysing the record of each of the current trainers in Smartform in order to produce, for each trainer with a runner today, the recent strike rate for that trainer. “Strike rate” is simply defined as the percentage runners to winners over a certain time period.

A trainer's strike rate is an interesting statistic for a number of reasons. For the time being we can assume that, as a result of backtesting, we have established that this is a potentially significant factor to use within the context of a betting strategy (in fact, in certain circumstances, betting on horses whose trainers have good strike rates can even be profitable in itself).

Producing a strike rate for each trainer is therefore a matter of querying the number of runs from each trainer over a certain period versus the number of wins for each trainer over the same time period. We can do most of this by remaining within the realm of SQL statements and querying the database to produce objects we will subseqently manipulate. The R commands for this are as follows:

```
# produce run count for each trainer over past 14 days

> runner_count_sql = paste("SELECT count(historic_runners.trainer_id) AS 'Runner
Count', historic_runners.trainer_name AS 'Trainer' FROM historic_races JOIN
historic_runners USING (race_id) WHERE meeting_date>'", Sys.Date()-14, "' AND
historic_runners.trainer_id IN (", ids, ") GROUP BY historic_runners.trainer_id",
sep="")
> trainer_runner_count=dbGetQuery(SF, runner_count_sql)

# produce win count for each trainer over past 14 days

> trainer_win_sql = paste("SELECT count(historic_runners.trainer_id) AS 'Win Count',
historic_runners.trainer_name AS 'Trainer' FROM historic_races JOIN historic_runners
USING (race_id) WHERE meeting_date>'", Sys.Date()-14, "' AND
historic_runners.trainer_id IN (", ids, ") AND finish_position=1 GROUP BY
historic_runners.trainer_id", sep="")
> trainer_win_count=dbGetQuery(SF, trainer_win_sql)

# create data frame for trainers who have had winners and add strike rate
# column to the data frame

> winning_trainers = merge(trainer_runner_count, trainer_win_count)
> winning_trainers$strike_rate=winning_trainers[,3]/winning_trainers[,2]
```

Now the data frame winning_trainers should contain data such as the below:

```
> winning_trainers
               Trainer Runner Count Win Count strike_rate
1             A Oliver           10         1  0.10000000
2          A P O'Brien           33         5  0.15151515
3          A W Carroll           20         2  0.10000000
4    Adrian McGuinness            9         1  0.11111111
5            B Ellison           24         1  0.04166667
6           B G Powell            6         1  0.16666667
7        B J Llewellyn            6         2  0.33333333
8         B M R Haslam            8         1  0.12500000
9         B S Rothwell            4         1  0.25000000
10            C F Swan            8         1  0.12500000
```

We can now reference the strike rate for any query on trainers running today, and by the same token rank all trainers within a given race according to their strike rate.

Indeed, betting on horses trained by trainers with a strike rate and a run count above a certain

percentage may provide substance for a profitable betting strategy in its own right. Let's suppose for a moment that it does, we can now merge all runners from today with their trainer strike rates as follows:

```
> daily_runners = paste("SELECT race_id, scheduled_time, course, name, trainer_name
AS 'Trainer', trainer_id from daily_races join daily_runners using (race_id) where
meeting_date=CURDATE() order by scheduled_time, course")
> runners=dbGetQuery(SF, daily_runners)
> runners_and_trainers=merge(runners,winning_trainers)
```

Betfair daily mapping

Smartform also maps all daily runner Ids and race Ids in the Smartform database to all Betfair runner Ids and race Ids using a daily Betfair mappings table, so we can get the daily runner Ids and race Ids which correspond to the Betfair Ids automatically.

First, let's run the query for the `betfair_daily_mappings` in Smartform, to produce the Smartform runner name and the Betfair Ids, as follows:

```
> bf_ids = paste("select scheduled_time, daily_races.course, daily_runners.name,
bf_race_id, bf_runner_id from daily_races join daily_runners using (race_id) join
daily_betfair_mappings using (race_id, runner_id) where meeting_date=CURDATE()")
> bf_runners=dbGetQuery(SF, bf_ids)
```

Now, we can merge these details with our existing data frame, giving a comprehensive set of strike rates, where these are positive, for all today's races and runners, as follows:

```
> bf_runners_and_trainers = merge(runners_and_trainers, bf_runners)
```

From this point, we can use the strike rates as an input to a model along with any number of other variables, in order to compute the "true" probability of each horse winning the race. Or, quite simply, we can apply some rules to bet the highest strike rates according to the distribution of other strike rates in any particular race.

The point with both approaches is that we can run this entire script automatically, as well as creating the bets to execute the strategy with the BetfaiR package, since we have the race Id and runner Id as part of the data set.

Appendix 1 - Access to Betfair API Services

The BetfaiR API is accessible to anyone with a Betfair account.

The basic level of access is the Free Access API, the product code for this is 82 (you use this in your`login` code as the argument to `api_access_type`, as detailed in Chapter 3).

The set of services available in the Betfair API is detailed in the official Betfair documentation Sporting Exchange API available at `http://bdp.betfair.com/index.php?option=com_weblinks&catid=59&Itemid=113`

Some services are restricted in use under the Free API. Generally these are services that enable high frequency access to markets or accessing multiple markets within a short timeframe. A full list of these services can be found at the Betfair Developer site `http://bdp.betfair.com`.

The Free level of access is fine for most betting robots. A discussion of the many viable strategies and techniques for automating betting using the free API is detailed in Automatic Exchange Betting - .

However, intensive use of the Betfair API, such as that required for some trading robots, may require a paid access subscription.

Nb. If you do sign up for a paid access subscription, you can take advantage of of an offer that Betwise agreed with Betfair for users of the betfaiR library which gives free access for 2 months. Just type “BetfaiR package” in the comments box when you sign up.

Appendix 2 - 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.

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

All the decimal odds available are reflected in the betfaiR package vector constant, `betfair_odds`. At the command line, type:

```
> betfair_odds
```

in order to get a screen printout for all 350 discrete decimal odds available.

You can use this vector to automatically get the next decimal price above the current one, or below, just as you would with any other vector in R. For example,

```
> which(betfair_odds==1.05)
```

will show us that the price 1.05 is the fifth value `[5]` in the vector `betfair_odds`. Let's say we have saved the current decimal odds in a vector `current_price`. So to get the next price available we simply have to add 1 to the return variable, as follows:

```
> current_price = 1.05
> current_price_position = which(betfair_odds==current_price)
> next_price = betfair_odds[current_price_position+1]
```