Python 3 Web Development

Beginner's Guide

Use Python to create, theme, and deploy unique web applications

Michel Anders

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Python 3 Web Development
Beginner's Guide

First published: May 2011

Production Reference: 1060511

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849513-74-6

www.packtpub.com

Cover Image by Rakesh Shejwal (shejwal.rakesh@gmail.com)

About the Author

Michel Anders, after his chemistry and physics studies where he spent more time on computer simulations than on real world experiments, the author found his real interests lay with IT and Internet technology, and worked as an IT manager for several different companies, including an Internet provider, a hospital, and a software development company.

After his initial exposure to Python as the built-in scripting language of Blender, the popular 3D modeling and rendering suite, the language became his tool of choice for many projects.

He lives happily in a small converted farm, with his partner, three cats, and twelve goats. This tranquil environment proved to be ideally suited to writing his first book, *Blender 2.49 Scripting* (*Packt Publishing*, 978-1-849510-40-0).

He loves to help people with Blender and Python-related questions and may be contacted as 'varkenvarken' at `http://www.blenderartists.org/` and maintains a blog on Python-specific subjects at `http://michelanders.blogspot.com/`.

For Clementine, always.

About the Reviewers

Michael Driscoll has been programming Python since the Spring of 2006 and has dabbled in other languages since the late nineties. He graduated from the University with a Bachelors of Science degree, majoring in Management Information Systems. Michael enjoys programming for fun and profit. His hobbies include Biblical apologetics, blogging about Python at `http://www.blog.pythonlibrary.org/`, and learning photography. Michael currently works for the local government, where he does programming with Python as much as possible. Michael was also a Technical Reviewer for *Python 3: Object Oriented Programming* by *Dusty Phillips* and *Python Graphics Cookbook* by *Mike Ohlson de Fine* (both by *Packt Publishing*).

I would like to thank my friends and family for their support and the fun times they share with me. Most of all, I want to thank Jesus for saving me from myself.

Róman Joost discovered open source software in 1997. He is the project manager for user documentation for GNU Image Manipulation Program (GIMP). Róman also helped with German internationalization of GIMP. He has been contributing to GIMP and Zope open source projects for eight years.

Róman has a Diplom-Informatiker (FH) from the University of Applied Sciences in Koethen (Anhalt). He has worked for Zope companies—Gocept GmbH & Co in Germany, Infrae in The Netherlands, and is currently working for a Zope company in Brisbane, Australia. For relaxation, he enjoys photography and digital painting with GIMP.

Tomi Juhola is a software development professional from Finland. He has a wide range of development experience from embedded systems to modern distributed enterprise systems in various roles such as tester, developer, consultant, and trainer.

Currently, he works in a financial company and shares this time between development lead duties and helping other projects to adopt Scrum and Agile methodologies. He likes to spend his free time with new interesting development languages and frameworks.

He has reviewed conference proposals, a Python development book, and has also published his own Master's theses on Agile embedded development.

Andrew Nicholson is a computer engineer with over fourteen years of professional experience in a broad range of computing technologies. He is currently a Technical Director with Infinite Recursion Pty Ltd.—a bespoke software engineering company located in Sydney, Australia. He is a passionate advocate and a participant in the free, libre, and open source software (FLOSS) community and has actively participated since 1999 contributing code, ideas, and energy in this engineering community. He was a Technical Reviewer for the book *Python Testing: Beginner's Guide* (2010), *Packt Publishing*.

Nicholson has a B.Eng (Computer) [Honours 1] from Newcastle University, Australia and a M.Eng (Wireless) with Merit from Sydney University, Australia.

Nicholson's biography can be read at `http://www.infiniterecursion.com.au/people/`.

Table of Contents

Preface

Building your own Python web applications provides you with the opportunity to have great functionality, with no restrictions. However, creating web applications with Python is not straightforward. Coupled with learning a new skill of developing web applications, you would normally have to learn how to work with a framework as well.

Python 3 Web Development Beginner's Guide shows you how to independently build your own web application that is easy to use, performs smoothly, and is themed to your taste—all without having to learn another web framework.

Web development can take time and is often fiddly to get right. This book will show you how to design and implement a complex program from start to finish. Each chapter looks at a different type of web application, meaning that you will learn about a wide variety of features and how to add them to your customized web application. You will also learn to implement jQuery into your web application to give it extra functionality. By using the right combination of a wide range of tools, you can have a fully functional, complex web application up and running in no time.

A practical guide to building and customizing your own Python web application, without the restriction of a pre-defined framework.

What this book covers

Chapter 1, *Choosing Your Tools*, looks at the many aspects of designing web applications. The idea is to provide you with an overview that may help you recognize components in subsequent chapters and give you some insight into the arguments used to decide which tool or library to use. We also illustrate some issues that are relevant when designing an application that does not deal with coding directly, such as security or usability.

Chapter 2, *Creating a Simple Spreadsheet*, develops a simple spreadsheet application. The spreadsheet functionality will be entirely implemented in JavaScript plus jQuery UI, but on the server-side, we will encounter the application server, CherryPy, for the first time and we will extend it with Python code to deliver the page that contains the spreadsheet application dynamically.

Chapter 3, *Tasklist I: Persistence*, a full fledged web application needs functionality to store information on the server and a way to identify different users. In this chapter, we address both issues as we develop a simple application to maintain lists of tasks.

Chapter 4, *Tasklist II: Databases and AJAX*, refactors the tasklist application developed in the previous chapter. We will use the SQLite database engine on the server to store items and will use jQuery's AJAX functionality to dynamically update the contents of the web application. On the presentation side, we will encounter jQuery UI's event system and will learn how to react on mouse clicks.

Chapter 5, *Entities and Relations*, most real life applications sport more than one entity and often many of these entities are related. Modeling these relations is one of the strong points of a relational database. In this chapter, we will develop a simple framework to manage these entities and use this framework to build an application to maintain lists of books for multiple users.

Chapter 6, *Building a Wiki*, develops a wiki application and in doing so we focus on two important concepts in building web applications. The first one is the design of the data layer. The wiki application is quite complex, and in this chapter, we try to see where the limitations in our simple framework lie. The second one is input validation. Any application that accepts input from all over the Internet should check the data it receives, and in this chapter, we look at both client-side and server-side input validation.

Chapter 7, Refactoring Code for Reuse, after doing a substantial bit of work, it is often a good idea to take a step back and look critically at your own work to see if things could have been done better. In this chapter, we look at ways to make the entity framework more generally useful and employ it to implement the books application a second time.

Chapter 8, *Managing Customer Relations*, there is more to an entity framework and CherryPy application code than merely browsing lists. The user must be able to add new instances and edit existing ones. This chapter is the start of the development of a CRM application that will be extended and refined in the final chapters.

Chapter 9, Creating Full-Fledged Webapps: Implementing Instances, focuses on the design and implementation of the user interface components to add and maintain entities, and relations between entities, in a way that is independent of the type of entity. This functionality is immediately put to use in the CRM application that we develop. Managing user privileges is another issue we encounter as we explore the concept of role-based access control.

Chapter 10, *Customizing the CRM Application*, is the final chapter and it extends our framework and thereby our CRM application by taking a look at browsing, filtering, and sorting large numbers of entities. We also take a look at what is needed to allow customization by the end user of the application's appearance and its functionality.

Appendix A, References to Resources, is a convenient overview of both Web and paper resources.

What you need for this book

Besides a computer running Windows or Linux to develop and test your applications, you will need the following pieces of open source software:

- Python 3.2
- CherryPy 3.2.0
- jQuery 1.4.4
- jQuery UI 1.8.6

How to obtain and install these packages is explained in detail in *Chapter 2*. We also use some additional plugins for jQuery and provide installation instructions where appropriate.

You will also need a web browser to interact with your applications. The applications were tested on Firefox 3 and Internet Explorer 8, but any moderately recent versions of these browsers will probably work just as well, as will Chrome. The Firebug extension for Firefox, however, is a superior tool to debug the client-side of web applications, so you might want to try it if you have not done so already. Appendix A provides links to the necessary resources.

Finally, you will need a text editor, preferably with syntax highlighting capabilities for Python as well as JavaScript and HTML. The author uses Notepad++ (`http://notepad-plus-plus.org/`) on Windows and JOE (`http://joe-editor.sourceforge.net/`) on Linux, but the choice is entirely up to you.

Who this book is for

Moderately experienced Python programmers who want to learn how to create fairly complex, database-driven, cross browser compatible web applications that are maintainable and look good, will find this book of most use. All applications in the book are developed in Python 3, but experience with Python 2.x is sufficient to understand all examples. JavaScript plays an important supporting role in many of the example applications and some introductory level knowledge of JavaScript might be useful, but is not strictly necessary because the focus is mainly on Python development and the JavaScript code is used either as building blocks or explained in such detail that anyone comfortable with Python should be able to understand it.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

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

Code words in text are shown as follows: "Running CherryPy's `setup.py` script installs a number of modules in Python's `Lib\site-packages` directory."

A block of code is set as follows:

```
<div id="main">
<ul>
<li>one</li>
<li class="highlight">two</li>
<li>three</li>
</ul>
```

```
</div>
<div id="footer">footer text</div>
```

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

```
t=t+'<thead class="ui-widget-header">
 <tr class="ui-helper-reset"><th></th>';
   for(i=0;i<opts.cols;i=i+1){
   t=t+'<th class="ui-helper-reset">' +
String.fromCharCode(65+i)+"</th>";
   }
```

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

```
python -c "import cherrypy"
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "New books or authors may be added by clicking the **Add new** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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

Customer support

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

Downloading the example code for this book

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

Errata

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

Piracy

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

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

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

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Choosing Your Tools

In this chapter, we look at the many aspects of designing web applications. The idea is to provide you with an overview that may help you recognize components in subsequent chapters and give you some insight into the arguments used to decide which tool or library to use.

Also, as this book covers more than just developing example applications, we illustrate some issues that are relevant when designing an application that does not deal with coding directly, like security or usability.

In this chapter, we will be:

- Indentifying the components that a web application consists of
- Choosing suitable tools
- Considering what designing for maintainability and usability implies

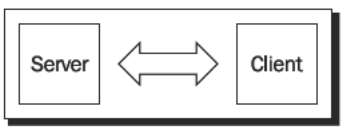
There is a lot of ground to cover, so let's get started.

Identifying the components of a web application

A web application is not a monolithic object. In designing such an application, it might help focus if you look at an application as a collection of related objects, each with its well-defined purpose. This can be done with multiple levels of detail and even the mile high view may already give some valuable insights.

Time for action – getting an overview of a web application

From the following picture shown, it should be clear that a web application is not a singular thing. It consists of parts that reside on a server and parts that run on the computer of the user. Both halves are just as important; although the server may hold the application data and implement the logic to modify that data following requests from the user, the data is displayed by the part of the web application running in the browser on the client computer and the user signals his/her request by interacting with the user interface components in the browser, for example, by clicking on an "OK" button.



- Think about your application and consider both server and client-side. The advantage of looking at the individual halves is that we might make choices that are optimal for the specific half.
- Look at the general requirements for the client half. For example, because we want to offer the user a sophisticated user interface, we opt for the jQuery UI library. This decision does not touch the overall design decision on the server, because apart from delivering the files that the jQuery UI library consists of, the choice of user interface library has no impact on the choice of the database engine or the server operating system for example.
- Look at the requirements for the server half. For example, consider which implementation language to use. We select Python as the language to implement the server-side code but if we had compelling arguments to switch to C#, we could do so without the need to change anything on the client.

If we zoom in on our web application, an image emerges of many interacting layers, each encapsulating a well defined piece of functionality. Everywhere two layers touch, information flows through a well defined interface (API). This helps in the separation of concepts (our application is only talking to the database layer to store and retrieve persistent data and only to the web server to return data upon request) but in practice, the separation between these layers isn't completely clear in all circumstances. For example, the server-side part of our application is actually an integral part of the web server.

This simple schematic of a web application is virtually identical to a regular client-server architecture. However, when we look more closely at the implementation of the client and the interaction between client and server, differences will emerge as we will see in the next section where we zoom in a bit closer.

What just happened?

With both halves of the application identified, we can now zoom in on each individual half.

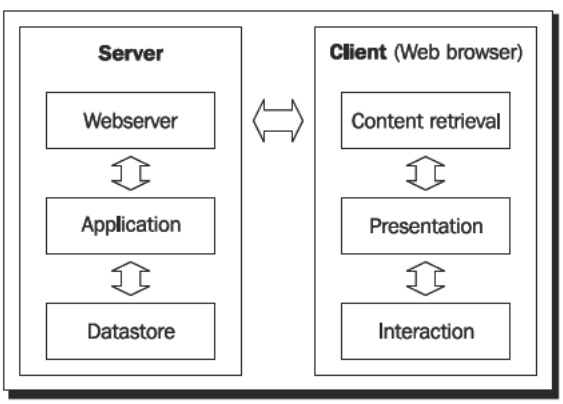
This will enable us to get a more detailed image, that will help us to make informed decisions regarding the smaller components that make up our application.

The main components are easy to identify:

- The data store holds data on the server (it is often a database engine, sometimes just files on the filesystem).
- The server-side application services requests that are passed through from the web server.
- The web server forwards those responses to the client again and may serve static files as well.

The web browser takes care of running the client side of the application, but within the browser, we can identify several layers of activities. These consist of:

- Fetching the content to structure the data (often HTML files)
- Running JavaScript code to enhance the presentation of the data
- Allowing interaction with the user



Of course we could zoom in even further to reveal additional detail like the operating system on the client and the server, or even the hardware and the network components and although occasionally useful, this would generally be overkill. With the main components clearly identified, we can take the next step and choose suitable tools to implement these components.

Choosing suitable tools

If you want to develop quality applications, you need suitable tools. Tools, of course, do not guarantee quality, but they can make life a lot easier. When developing web applications, there are two kinds of tools you need to consider: the ones you use to design, build, test, and deploy your application, like editors, version management systems, test frameworks, and maybe a package tool, and the tools that deliver your application to the end user. That last set of tools consists of a whole chain of components, from server operating system, web server, and database engine, all the way to the web browser and the JavaScript libraries used to display and interact with the application.

When we start a project, we have to know which tools we need and have to understand the capabilities and limitations of the many variations of these tools. There are, for example, quite a few JavaScript libraries that may be used to provide cross-browser compatible user interaction.

The trick is to make an informed choice. These choices are not necessarily limited to open source tools. If budget permits, it might be worthwhile to have the benefit of the special features many commercial development tools and libraries offer, but in this book, we limit ourselves to open source and/or free resources. This makes sense as the cost of tooling and licenses in small projects can make a significant dent in a budget.

The opportunity to use free tools might not exist for the deployment environment. You may well develop your application on your own Linux box, but test and deploy it on a Windows server. The latter needs a license that will not be free, but even open source options are not always free. Many companies nowadays shift to deploying their applications to the cloud and even though these machines might be running an open source operating system, you pay not only for CPU power and bandwidth but also for support, the latter being crucial in applications that will lose you money if they are not running. However, using open source tools in general gives you a much wider choice because many tools run equally well on any platform.

In the following sections, we will look at the many components that make up the tool chain and will try to show what arguments were used for the choices made for developing the applications in this book and what (if any) viable alternatives are there. Note that some arguments are quite subjective and the choice finally made does not necessarily indicate that the alternative is bad; we certainly are not attempting to start flame wars over which tool is better. We simply list requirements for application development as we see it and try to find the tools suitable for the task. In some situations, another tool might be better, but for this book, we try to find a matching toolset that can be used for all sample applications that are free (as in beer) and easy to learn and use.

Time for action – choosing a delivery framework, also known as web server

In the first section of this chapter, we showed that a web application lives in two realms at the same time, namely, on the server and on the client. In order to deliver information to the client and receive a response in return, our web application needs two important items at the server: a delivery framework and an application to compose content and respond to the request.

The delivery framework might be a full-fledged general purpose web server such as Apache or Microsoft Information Server, but although these are very versatile and come with many options to tune the web server to your specific needs, they certainly take quite some time to get acquainted with and it takes extra attention to integrate the dynamic content of your application with these servers. If performance is crucial or the requirements for your project include that your application has to be deployed as part of these servers, you may not have a choice, but otherwise its worth looking at the alternatives that are simpler to use or offer integration advantages.

So what do we need?

- A fairly lightweight web server that is easy to configure and maintain
- That allows for smooth integration of static and dynamic content
- That comes with reusable components that ease the development process
- That is actively maintained and developed

Given these requirements, our choice for delivery framework is CherryPy.

What just happened?

CherryPy fits the bill nicely. Its main advantages are:

- CherryPy is written in Python and components that deliver dynamic content are written as Python classes that are tightly integrated with CherryPy's core.
- CherryPy comes with a whole host of **tools**; reusable components that can be used to implement anything from custom error pages to session management.
- CherryPy has a proven track record as the core web server of the larger TurboGears network.
- And finally, CherryPy is actively developed and enjoys a large user community.

The disadvantage of being written in Python is that performance might not be top notch, but we will look into that in the next section.

Time for action – choosing a server-side scripting language

When developing web applications, you have a virtually unlimited choice of programming languages you can use, so we have to consider what is important for us in our project and make a tradeoff if necessary.

- Consider how important development time is compared to performance. Compiled languages like C# or C++ might be used if CPU power is scarce or if you do not want to distribute the source code in a readable format. But when development time is at a premium, using scripting languages often saves time as they make it easier to develop applications in an incremental way, even to the point where you can type in commands interactively to see what is possible and later incorporate these trials in your code.

  Performance is generally not an issue, especially when using scripting languages that are compiled to intermediate byte code, as is the case for languages like Python and Perl, for example. And while it is true that scripted languages are compiled each time they are run, this has a negligible effect when the program is a long running web application.

- Weigh the importance of debugging. Interpreted languages are often simpler to debug as compiled languages, both because the debugger has access to more information that you may explore interactively if something breaks and because you can try out any modules you have written by interactively calling functions to see what happens.

- Think beyond the project. Once implemented and deployed, your application might have a long and happy life, but that inevitably means that there will be requests for smaller or larger changes and choosing a suitable language can help to reduce the maintenance effort. Compared to compiled languages that in general have quite low-level instructions and language constructs, interpreted languages have (very) high level constructs that make for condensed code that packs a lot of meaning in a few statements. That is not only easier to read but also faster to interpret and in the end these high level constructs, once interpreted, run at (almost) compiled speed making the performance difference sometimes hard to spot. More meaning and less code do make for easier reading and this is a huge benefit when maintaining code.

In the end, the choice for the language to implement the web application is at least in part a matter of taste, but in this book we opt for Python as it offers an optimal tradeoff between the different considerations.

What just happened?

Now that we have chosen Python as our server-side scripting language, let's have a good look at the arguments:

- Python is easy to read and therefore easy to learn and maintain. Although Python is relatively unique among programming languages in treating whitespace as meaningful in many places, this does enhance readability quite a lot.
- Python is a very high level language, incorporating concepts like list comprehension and functional programming. This allows for compact programs that pack a lot of functionality in little code, enhancing readability and reducing maintenance.
- Python comes "batteries included". Python is distributed with a vast amount of well designed and well maintained libraries (modules) that provide anything from access to `.csv` files and parsing XML, to building an HTTP server with a handful of code and these modules are at least as well documented as the language itself. This all means we can cut down on development time as in many cases we do not have to reinvent the wheel ourselves.
- Python has many third party modules. Even if a module is not included with the distribution, chances are somebody has written just the module you are looking for.
- Python is an object-oriented language. This is widely regarded as a good thing as it aids in data abstraction but its main attraction to people developing database-driven applications is that it allows for a natural way of mapping tables to types (classes). Records in a table of cars can be mapped to a 'Car' class and instances of this class can then be manipulated in much the same way as native classes like strings or lists. This again makes it easier to read the code and therefore maintain the code.
- Python is available on many cloud platforms. To run a Python program on the server, you need Python deployed on that server. If you have full access, this might not be an issue and indeed hosting companies provide (virtual) machines with Python already installed but for very lightweight cloud platforms like Google Gears, your choice of available languages might be limited. However, Python (together with Java) is fully supported by Google Gears and although this is not a consideration for the example applications in this book, it might be for your applications.

The version of Python we use in this book is version 3 (version 3.2 at the time of writing). Although not all third party modules are (yet) ported to this new version, it is the best version to use if you want to develop in a future proof way.

Python's multi-threading capabilities at the moment do not allow for optimal usage of multi-core processors. Most implementations of Python do not allow running separate threads truly in parallel. This is by far not as bad as you may think though, as this restriction is mainly valid for interpreted python code, not necessarily for code running in, for example, the OS kernel. And because in a web server a lot of time is spent waiting for packets to be sent or received over the network, this mostly does not affect the performance of your Python code. In the future, the multi-threading implementation of Python may change, but this is a hotly debated subject. More on this subject can be found by searching for "Python 3 GIL".

Time for action – choosing a database engine

One of the key requirements of any web-application is that is has access to some sort of persistent storage. This might be used to store core data like a catalog of car parts, but a password file also needs a form of persistent storage.

Often it is possible to store the information in files on the filesystem and indeed some of the applications we develop in this book do just that, but if you have a lot of structured data or you find that many people want to access this data at the same time, it is usually a better choice to store this data in a database and access this data through a database engine.

When choosing a database engine, you should consider the following points:

- Does it offer the functionality you need? Database engines are sophisticated pieces of software and in general offer a lot of functionality, often more than you need. Although this may sound like an advantage, all these features must be learned by a developer to take advantage of them and may complicate your code which may increase the effort to maintain an application.
- Is it easy to install and maintain? Database engines often run as separate applications that are accessed over a network. This means that they have to be installed, tested, and maintained separately. This may add significantly to the effort needed to deploy your application. And installation isn't even everything; you will have to consider operational issues as well, for example, how much effort it is to set up a suitable backup scheme or how to monitor the availability of the database.
- Does it offer an API that is simple to use from your chosen programming language and does this API provide access to all necessary functionality?
- And finally, does it perform well enough to respond swiftly to the requests of your application, even during peaks?

Python offers a standardized API to access many available database engines, including MySQL and PostgreSQL. Fully in line with its 'batteries included' philosophy, Python also comes included with a database engine and a module to access it. This database is called SQLite and is a so called embedded database: it doesn't run as a standalone process that can be accessed through some means of inter-process communication, but the database engine is an integral part of the program that uses it. Its only external part is a single file containing the data in the database itself and that may be shared by other programs that include the SQLite engine. As it fits our requirements, SQLite will be the database engine we will use for the applications we develop in this book.

What just happened?

Our choice for SQLite as the database for many of our applications is easily justified:

- Although not as feature-rich as, for example, MySQL, it does provide the functionality we need.
- Installation is practically a no brainer as SQLite comes included with Python.
- The API offered by the `sqlite3` module gives access to all functionality.
- It performs well enough for our needs (although statements about performance are very difficult to make in advance).

The main arguments supporting the use of SQLite in our applications are not its speed, small memory footprint, or reliability (although these are certainly not drawbacks as SQLite's reputation as database engine of choice for mobile telephone appliances proves) but the fact that because it is embedded in your program, it obviates the need for a separately configured and maintained database engine. This cuts down on maintenance in a serious manner as database engines are demanding beasts that take a lot of care and feeding. Also, because it is included in Python, it reduces the number of external dependencies when deploying an application.

A final argument is its type system that closely resembles Python's type system; in contrast to many other database engines, SQLite allows you to store any value in a column no matter how this column was typed when it was created, just like you can store a string in a Python variable that was first used to store an integer value. This close correspondence of types allows for an intuitive mapping of Python values to values stored in the database, an advantage that we will study closely when we encounter our first application that uses SQLite.

The integration with Python is so close that it is possible to use Python functions within the SQL expressions used to query SQLite. The native set of functions in SQLite is quite small compared to other database engines but the ability to use Python functions removes this limitation completely. It is, for example, straightforward to add a hash function from Python's `hashlib` module, that is very convenient when implementing a password database.

Time for action – deciding on object relational mappers

Relational database engines like SQLite use tables consisting of rows and columns as their primary data abstraction model. Object-oriented languages like Python define classes to instantiate objects that have attributes. There is a fair amount of correspondence between these concepts as class definitions mimic table definitions where object instances with attributes relate to records with columns but maintaining the integrity of that relation is not so straightforward.

The problem not only lies in the different languages used to define tables and classes. The main issue in relational databases is maintaining referential integrity. If you have, for example, a record representing a car part that references a record in a different table that represents a car type, then a relational database lets you define explicit actions to execute if, for example, the record representing the car type is deleted. These constraints are of course possible to implement in Python data structures as well, but it does take serious effort to implement.

Finally, most database engines require fixed data types for each column whereas Python variables and attributes may refer to any kind of data type. This restriction is not present in SQLite but even SQLite cannot store everything without conversion. A Python variable, for example, may refer to a list of objects, something that cannot be stored in a single column of a relational database.

Still, we would very much like to have a way to store object instances in a relational database or at least the data contained in those object instances, and have the means to define the relation between classes and tables in a maintainable way. To this end, many people have designed solutions in the form of object relational mappers. For Python, quite a few exist that are both mature and robust tools (like SQLAlchemy).

When deciding which tool to use, you should at least consider the following:

- How much time it will cost to learn to use it. Those tools are usually very versatile and quite often require a considerable amount of effort to learn.
- How will it affect development and maintenance? Complex tools may help to solve the challenge of creating an effective and efficient mapping between classes and tables, but may require an idiom that detracts from a clear overview of your implementation. This may well be worth it, if your data model consists of many classes and performance is an important consideration, but for smaller projects the added complexity might be too great of a disadvantage when it impacts significantly on the development time.

Because the focus in this book is on understanding the choices in implementing web applications and persistent storage, using a complex tool like an object relational mapper may hide all kinds of aspects necessary to gain understanding.

Therefore, we will not use a third party object relational mapper in the examples in this book but implement increasingly versatile storage solutions in each chapter, tackling specific requirements as we encounter them. We will see that in many situations an object relational mapper is superfluous, but in the final chapters, we will build a simple framework ourselves to give us not only a tool but an insight into the intricacies of mapping complex assemblies of classes to tables in a database as well.

Time for action – choosing a presentation framework

Web applications might be all about accessing and manipulating data from within a web browser but the way the application looks and feels to the user is just as important. A user interface that is non-intuitive, sluggish, or fails to work on some mainstream browser will not invite users to use your application again.

HTML, the markup language commonly used to display content, does allow for some interaction through the use of `<form>` elements and the way a page is presented can be styled with cascading style sheets, but its use has some major drawbacks:

- It is quite difficult to create user interface components from basic building blocks that resemble commonly used applications.
- The use of HTML feels sluggish because each form, when submitted, fetches a completely new page.

Fortunately, all major browsers support JavaScript and that language can be used to add a whole new level of interactivity. However, in order to smooth out all inconsistencies between browsers, you can save a lot of development time when you use a JavaScript library that takes care of those inconsistencies and adds cross browser compatible user interface components (widgets).

Although such libraries are used client side, HTML pages can be composed in a way that instructs the browser to fetch these libraries from a central source, for example, the same server that serves the web application. This way, the use of these libraries imposes no extra requirements on the browser.

Some points to consider when choosing a suitable library are:

- Is it really cross browser compatible? Not all libraries support each and every browser. This might be important if your application still needs to work with a fairly old browser.
- Does it offer the graphical components and functionality you need?
- Is it well designed and documented, extensible, and consistently implemented? After all, such a library should be fairly easy to learn and as no library can offer everything, extensibility and especially how easy it is to extend it are important considerations.
- Does it have an active user community? All the more important here because such a community may not only provide answers to your questions, but may be a good source of reusable components.

Based on these considerations, we choose to use two intimately connected JavaScript libraries: jQuery and jQuery UI.

What just happened?

Let's have a look at why jQuery and jQuery UI are such a good choice.

jQuery provides the functionality to select and manipulate HTML elements on a page and jQuery UI provides a number of sophisticated widgets and effects. Together, they offer many advantages:

- jQuery not only hides browser inconsistencies, but its methods take CSS3 compatible selectors even on browsers that do not support CSS3 in the style sheet they accept.
- Both libraries are widely used, actively maintained, free, and are distributed as small files. The latter is important when you consider that these files need to be transferred from server to client so any bandwidth saved is good.
- jQuery UI offers a rich set of well designed and professional looking graphical components and effects.

Other advantages of the wide adoption of these libraries are that there are many resources available to get you started and that many people have written plugins that extend the usability of these libraries even more. As we will see on many occasions, the essence of developing a good application efficiently is often choosing the right plugin for the job.

Designing for maintainability and usability

It is one thing to come up with a great idea on how to implement some web application but yet another to design an application in such a way that it will be easy to maintain and use. Designing with these considerations in mind will make all the difference between a professional application and a mediocre one.

Testing

Everybody will agree that it makes sense to test an application before it is deployed but thorough testing requires some serious effort. Testing is also often considered as boring or even detracting from the 'real' development work and shares this aura with writing documentation.

However, testing gives you a better feel for the quality of the application you deliver and a test framework, however simple, is always better than none, especially for the kind of small to medium web applications we look at in this book, as these tend to be written by very small teams that quickly prototype and often change the code as insight progresses and customer requirements change. Having a test suite at hand ensures that at least the parts of the code that don't change keep on performing as expected.

Of course, not everything can be tested and the tools needed to test part of your code should be simple to use, otherwise there is no incentive to keep on using them. We will look at **unit tests** for a number of modules we develop in Python. Unit testing is an approach where we try to define the behavior of an isolated piece of code (for example, a single method) and check whether this code produces the expected results. If the implementation of the code changes but the tests still show no failure, we know that the new implementation can be used safely.

Time for action – choosing a test framework

When choosing a test framework, ask yourself the following questions:

- What do I want to test? You cannot test everything and developing tests takes time.
- How easy is it to write and maintain the tests? This question is just as relevant for developing tests as it is for developing code in general.
- How much effort is needed to perform the tests? If it is easy to automate the tests, they can, for example, be run as part of the deployment as an extra check.

Just for Python alone there are quite a few testing frameworks available, but we will choose the `unittest` module distributed with Python. Note that although we choose to write only automated test for the Python parts of the applications, this doesn't mean we have not tested the JavaScript parts, but user interactions tend to lend themselves less to an automated way of testing so we do not address that in this book.

What just happened?

For the Python unit tests, we restrict ourselves to the `unittest` module that is distributed with Python, as this will not introduce any new dependencies on external tools but also because:

- It is fairly simple to learn and use.
- It produces clear messages if a test fails.
- It is easy to automate and may easily be integrated with, for example, a setup script.

Version management

A version management tool is normally not part of a web application and not strictly required to develop one. However, when you want to keep track of changes in your code, especially when the number of files keeps on growing, a version management tool is invaluable.

Most come with integrated functionality to show the differences between versions and all have the possibility to annotate a version or revision in order to clearly mark it. Widely used open source solutions are **git** and **svn**.

Both may operate as a server that can be accessed through a web browser but command-line tools are available as well and svn even has a very user-friendly integration within Windows' file explorer. Both have their strengths and weaknesses and it is hard to declare a clear winner. This book and its accompanying examples were all maintained in svn, primarily because of the ease of use of the Windows client.

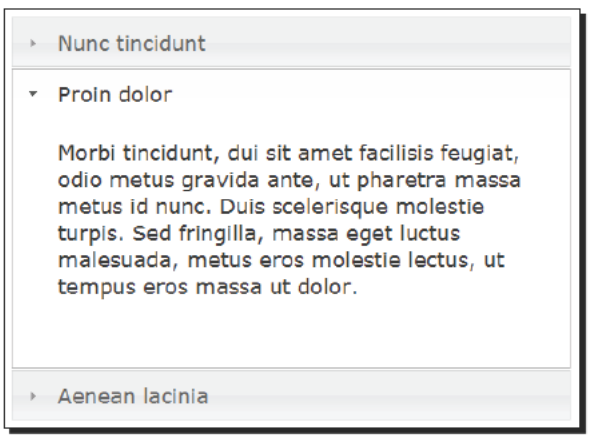
Usability

Web applications are built for end users, not for developers. It is not always easy to design an interface that is easy to use. In fact, designing really good interfaces is difficult and takes considerable skill and knowledge. However, this does not mean that there aren't any rules of thumb that can help you prevent usability disasters. We look at some of them in the following sections.

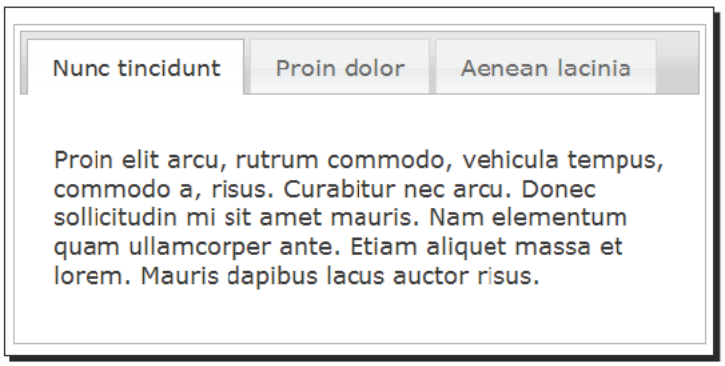
Good looking – adhering to common GUI paradigms

Applications are easier to use if the interface components are already familiar. Therefore, it is generally a good idea to look at applications that are successful and used by many people.

A common concern in many applications is the need to present a lot of information in a small amount of space. It is therefore no wonder that many modern applications use accordion menus and/or a tabbed interface to structure that data, such as the following screenshots:



An accordion menu is great for displaying a fair amount of information in a side bar but even more information can be presented in tabs:



Examples are found in all recent editions of common office productivity software, web browser, and CRM applications. Having a good look at the ones you like working with yourself might be a good start. In the larger applications developed in this book, we will certainly refer to some key applications that may be used as an inspiration.

Themable

Choosing a consistent and pleasing color scheme and font makes an application more coherent and therefore more pleasurable to use. An overload of information can baffle people and using a wild color scheme or many different fonts will not help in getting an overview of the data that is presented.

But whether your user interface supports the concept of a theme that is easy to change plays an important role in other areas as well. You probably want your web application to blend in well with the rest of your website or to convey some sort of company or brand identity. Using a consistent color scheme will help. It might even be desirable to offer a choice of themes to the end user, for example, to provide people with visual impairments with high contrast themes for better legibility. The library fully supports the use of themes and makes it simple to extend this themability to widgets we design ourselves.

Cross-browser compatible

Web applications are often geared to a specific audience, so it might be possible that the requirements specify only a single browser, but in general, we don't want to deny the user his/her favorite browser. jQuery takes away most of the pain in supporting more than one browser. Our apps are designed for Internet Explorer 8, Firefox 3.x, and Google Chrome, but probably will run on most other browsers as well. Note that 'probably' might not be good enough and it is always a good idea to test your application specifically on any required platform!

Cross-platform compatible

Client-side, the web browser is the key component in our chain to watch out for and therefore, the operating system it is running on will quite likely not be a source of problems.

Server-side, we want to keep our options open as well. Fortunately, Python is a cross platform solution, so any Python program that runs on Windows will normally run on GNU/Linux as well and vice versa.

We should be careful though when using modules that are not distributed with Python and are not pure Python. These might be available on every platform but it is better to check beforehand. The applications in this book use only modules provided in the standard Python distribution, with the exception of CherryPy, which is a pure Python module and should run on every platform.

Maintainability

Writing code is hard work, maintaining it can be even harder. We briefly touched upon this subject earlier when we discussed the use of a testing framework, but maintaining code is more than being able to test it.

Standards compliant

An important concept in creating code that is easy to maintain is being standards compliant. Adhering to standards means that other people stand a greater chance in understanding your code.

SQL, for example, is a query language that most database engines understand. Therefore, it is less relevant which engine we use for people maintaining the code as they do not have to learn an obscure query language.

Another example is communication between client and server. We can devise our own protocol to construct requests in JavaScript and respond to those requests in Python, but it is a lot less error prone to use documented standards like AJAX to communicate and JSON to encode data. It also saves on documentation as people can be referred to any number of books, if they want to learn more about those standards.

Standard does not necessarily mean 'approved by some independent organization'. Many standards are informal but work because everybody uses them and writes about them. Both AJAX and JSON are examples of that. Also the Python programming language is a de facto standard but JavaScript enjoys a formal standard (which doesn't mean all implementations adhere to the standard).

Security

Security is often regarded as an obscure or arcane subject, but security covers many practical issues that play a role in even the smallest web application. We wouldn't want anyone to access a paid-for web application, for example. However, security is more than just access control and we touch briefly on some aspects of security in the next sections.

Reliable

A web application should be reliable in its use. Nothing is more annoying than being presented with a server-side error halfway in the process of filling in a mortgage application, for example. As a developer and tester, you take care of testing the software thoroughly in the hope of catching any bugs but before implementing the application, the reliability of the software and libraries it uses should be taken into consideration.

You should especially be wary of using the latest and greatest nifty feature of some library in production software. This might be fun when whipping up some mock up or concept application, but do ask yourself if your customer really needs this bleeding edge feature and if he's/she's not better off with a tried and tested version.

Many open source projects (including Python) develop and maintain both a so called stable branch and a development branch to show off new features. The former you should use in production applications and the latter should be tried elsewhere.

Robust

Applications should not only be as bug-free as possible, but should also perform nicely under stress as well. The performance should be as high as possible under load, but just as important you should know what to expect when the load reaches some threshold.

Unfortunately, tuning for performance is one of the trickiest jobs imaginable because all components in the chain may play a role. Server-side considerations are the performance of the database engine used, the scripting language, and the web server.

Client-side, the quality of the presentation framework and the overall performance of the web browser are important and in between the server and client is the great unknown of the characteristics of the underlying network.

With so many variables, it is not easy to design an optimal solution in advance. However, we can test the performance of individual components and see if the component is a bottle neck. For example, if it takes three seconds to refresh a page provided by a web application you can rule out the database engine as a bottleneck if you can time the database access independently. The knowledge gained creating unit tests can be reused here because we already know how to isolate some functionality, and adding a timer and asserting that the response for a query is fast enough can be made a test itself.

It is also quite feasible to separately measure the time it takes to fetch a web component and to render it in the browser with a tool like Firebug and get an idea whether the client or the server is the bottleneck. (Firebug is a Firefox extension and can be found at `http://getfirebug.com/`).

Access control and authentication

In almost every application that we develop in this book, we implement some sort of authentication scheme. Most of the time, we will use a simple username/password combination to verify that the user is who he/she claims to be. Once the user is authenticated, we can then decide to serve only certain information, for example, just a list of the tasks belonging to him/her, but no tasks of any other user.

However, whether access to information is allowed, isn't always that basic. Even in simple applications, there might be a user who should be allowed more than others, for example, adding new users or resetting passwords. If the number of different things a user is allowed to do is small, this is straightforward to implement, but if the situation is more complex, it is not that easy to implement, let alone to maintain.

In the more elaborate applications featured in the later chapters of this book, we will therefore adopt the concept of role based access. The idea is to define roles that describe which actions are allowed when assuming a role. In a customer relations management application, for example, there might be three roles: a sales person, who is only allowed to access information for his customers, the sales manager who may access all information, and an administrator who may not access any information, but is allowed to back up and restore information, for example.

Once the rights of these roles are clear, we can associate any or all of these roles with specific persons. A small organization, for example, may have a technically savvy sales person who can also assume the admin role, yet still be unable to access information about customers other than his own this way.

If rights associated with a certain role are changed, we do not have to repeat this information for each and every person that may assume that role, thus making administration that much simpler.

Confidentiality

In some applications, we may want to make sure no one is listening in on the data transferred between the browser and web server. After all, in general you do not know which path your data takes, as it is routed across the Internet and at any point there might be someone who can intercept your data.

The easiest way to ensure confidentiality is to use connection level encryption and the HTTPS protocol does just that. The web server we use, CherryPy, is certainly capable of serving requests over HTTPS and configuring it to do so is quite simple but it involves creating signed certificates which is a bit out of the scope of this book. Refer to `http://www.cherrypy.org/wiki/ServerObject` for more information.

Integrity

The last aspect of security we talk about in this context is data integrity. Corruption of data may not always be prevented, but wholesale destruction may be guarded against with proper backup and restore protocols.

However, data corruption lurks in very small corners too. One of the trickiest things that can happen is the possibility of inserting data that is wrong. For example, if it is possible to input a date with a month outside the range 1-12, very strange things might happen if the application relies elsewhere on dates having the correct format.

It is, therefore, important to prevent the user entering wrong data by building in some sort of client-side validation. An excellent example is jQuery UI's `datepicker` widget that we will encounter in *Chapter 3*, *Tasklist I: Persistence*. If a text input field is adorned with a `datepicker`, the user can only enter dates by selecting dates from the `datepicker`. This is a great aid to the end-user, but we should never rely on client-side validation because our client-side validation might be inadequate (because it contains a bug or doesn't check all cases) and certainly cannot prevent malicious users from connecting to the server and actively inserting wrong data. We do need server-side input validation as well to prevent this and we will encounter some examples of it.

The key thing is to provide both: server-side validation as a last resort and client-side as an aid to the user.

A final word on security

Security is complex and tricky and details may be overlooked easily. You might know you have a front door made of 10 centimeter oak with state of the art steel locks, but if you forget to lock the backdoor all that oak and steel serves no purpose. Of all the subjects touched upon in this book, security is the one that you should always talk over with an expert. Even an expert cannot give you guarantees but taking a fresh look at the security requirements might keep you out of trouble. Make sure that you run the sample applications provided in this book in a secure environment behind a well managed firewall.

Help, I am confused!

Reading this chapter, you may get the feeling that developing web applications is horribly complex, even if you use the right tools. So many things may play a role! Do not despair though.

Time for action – maintaining overview

If you take a close look, you will see that none of it is rocket science, the most it takes is common sense and attention for detail, and in every chapter, we highlight the relevant issues in a straightforward language where it is relevant. Remember that this is a practical book, there will be many working examples that are examined in close detail and we won't inundate you with theory, but give you just enough to get things done.

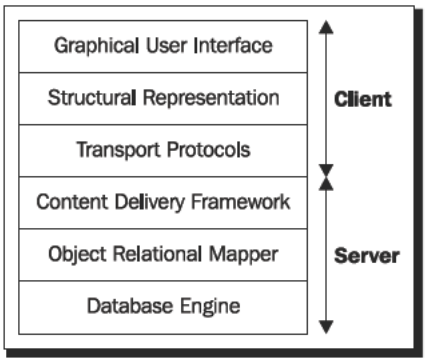
At every step in the development process, ask yourself the following questions?

- What needs to be done? There is no need to work on all things at the same time, indeed this is practically impossible. Allow yourself to form a high level idea first and identify the components in the next level down. Don't get bogged down with details when the outline is not clear yet.
- Which components of the application are involved? Identify the specific components involved when you develop a piece of functionality. The whole idea of identifying layers and components is to be able concentrate on a limited part of the application when developing.

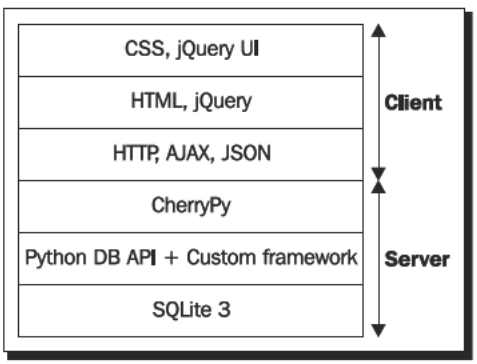
  This might not always work perfectly, but it certainly helps in maintaining focus. For example, when developing parts of the presentation layer, you may find that additional content is needed that should be provided by the delivery layer. Instead of immediately switching focus to that delivery layer, it's often simpler to define what is needed and complete the part of the presentation layer you are working on.
- What are the requirements? There is no need to implement stuff that is not needed. This may sound obvious, but many developers nevertheless fall into this trap. It is tempting of course to design your code to be as flexible as possible but it takes a lot of time, and as requirements change, it is unlikely that it'll prove flexible enough. Instead, the effort is better spent on writing code that is easy to understand so that the inevitable changes in requirements take less time to process.

What just happened?

When asking those questions and given the choices we made in this chapter, it might be helpful to draw a new picture that illustrates the technologies we will use:



The different components between the server and client that together make up the web application can be pictured as a layered stack. For each layer, we have chosen one or a few technologies, as illustrated in the following diagram:



Each application we encounter will be based on this model, so it might help to refer to this diagram once in a while if you feel you have lost track.

After reading this book, you will be left with the feeling that writing good, useable web applications is maybe a little bit more involved than you might have thought at first, but that is certainly within the reach of even the smallest of teams. Armed with all the fresh knowledge and practical experience, you will not have to compromise on quality, not even in the smallest project.

Summary

This chapter gave us a head start in providing an overview of the components and techniques involved in creating a quality web application. Specifically, we looked at:

- The components that make up a web application.
- The technologies we choose to implement these components.
- Which other issues play a role in the design, like security and usability.

With this extra knowledge, nothing can hold us back from writing our first web application in Python and that is exactly what we will do in the next chapter.

2
Creating a Simple Spreadsheet

In this chapter, we will develop a simple spreadsheet application. The spreadsheet functionality will be entirely implemented in JavaScript plus jQuery UI, but we will configure CherryPy to deliver the page that contains the spreadsheet application dynamically.

On the presentation side, we will encounter our first jQuery UI widgets (buttons) and will see how we can design other elements to adhere to jQuery UI standards to fit seamlessly in jQuery UI's theme framework. We will also see how to find and use publically available jQuery plugins and integrate the jEditable plugin into our application.

That is a lot to grasp in one go, but don't worry if every detail is not clear the first time. We will encounter many variants of the issues first encountered here in the other chapters and will explain all relevant details again in their context.

In this chapter, we will be:

- Creating an environment to develop and deliver our applications
- Designing a simple spreadsheet application
- Learning how to configure CherryPy to deliver this application
- Encountering our first jQuery UI widgets
- And designing our own jQuery plugins

There is a lot of ground to cover so let's go...

Python 3

Python 3 is the language we will use to develop the server-side parts of our applications. At the time of writing, the current stable version is 3.2. Installers and source archives are available for various platforms (including Windows, Mac OS, and Linux distributions) and may be downloaded from `http://www.python.org/download/`.

Time for action – installing Python 3

Downloading and installing Python is not difficult. The installers for many platforms can be downloaded from `http://www.python.org/download/`.

1. Download the installer for your platform and follow the installation instructions.
2. Verify that you correctly installed Python by typing the following command on the command line (for example, inside a Windows command prompt or a Linux xterm):

   ```
   >python --version
   ```

   The response will be the version:

   ```
   Python 3.2
   ```

What just happened?

On a UNIX-like system (like Ubuntu Linux, or Mac OS), Python might very well be already installed, so it might be a good idea to verify that first by trying the instructions in step 2. If the version returned is lower than 3, you should update your Python distribution. Note that it is perfectly possible to install version 3.x of Python alongside a 2.x version in order to not break applications depending on version 2.x (Python 3 is not backward compatible with version 2).

CherryPy

Writing an HTTP server in Python isn't that difficult, but writing and maintaining a robust and fully fledged web server that can act as an application server is quite something else. As we explained in *Chapter 1*, *Choosing Your Tools*, we will use CherryPy as our application server. At the time of writing, CherryPy's latest stable version for Python 3 is version 3.2.0 and can be downloaded from `http://download.cherrypy.org/cherrypy/3.2.0/`.

Windows users should use the zip archive and unpack it before proceeding to the instructions in the next section. There is also a `msi` installer available at the indicated location, but this installer might not be able to find the correct Python installation in the Windows registry and will only work on 32-bit versions of Windows. Unpacking the zip archive and following the setup instructions next is therefore a safer bet and also identical on both Windows and Unix-like platforms.

Time for action – installing CherryPy

The one thing to be careful with when you install CherryPy is that you have to make sure you install it in the right directory if you have more than one Python version on your system. CherryPy uses a setup script to install itself and one way to make sure the CherryPy modules end up in the correct place is by invoking Python explicitly with a full path, for example:

```
cd C:\CherryPy-3.2.0rc1
c:\Python32\python.exe setup.py install
```

What just happened?

Running CherryPy's `setup.py` script installs a number of modules in Python's `Lib\site-packages` directory. You may verify this was successful by typing the following on the command line:

```
python -c "import cherrypy"
```

This checks whether we can import the `cherrypy` module. If everything is installed correctly, there will be no output produced by this command. However, if CherryPy isn't installed, this may be signaled by an error message:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named cherrypy
```

When you have more than one version of Python installed, be careful to enter the complete path of the Python executable to select the correct version, for example:

```
C:\python32\python -c "import cherrypy"
```

Installing jQuery and jQuery UI

The applications we will design and implement in this chapter and the following chapters depend heavily on the jQuery and jQuery UI libraries. These libraries consist mainly of JavaScript files and some cascading style sheets, and images to style the widgets.

These files are served to the web browser as part of the application, and in general, there are two locations where they may be served from:

1. A (sub)directory on the server that runs CherryPy together with the other files that make up our application.
2. Or from an external web location like Google's or Microsoft's content delivery networks.

The latter option might be the best choice if your application gets a lot of traffic as these publicly available resources are designed for high availability and can cope with an enormous number of requests. This might seriously reduce the load on your server and thus reduce costs. More information on this can be found on jQuery's download section `http://docs.jquery.com/Downloading_jQuery#CDN_Hosted_jQuery`.

For development purposes it is often better to download the necessary files and serve them from the web server that serves the rest of the application as well. This way we can inspect those files easily when some error occurs or even decide to tweak the contents. If we choose to theme our application in a customized way (see the info box on jQuery UI's themeroller), the cascading style sheets will differ from the standard ones so we will have to serve them from our web server anyway.

In the example code provided with this book, we include both the jQuery and jQuery UI libraries in the `static` subdirectory of the directory for each chapter. There is also a `css` subdirectory that contains a set of customized style sheets that are optimized to deliver a visual style that is well readable both in print and onscreen. The version of the jQuery library used in this book is downloaded from `http://code.jquery.com/jquery-1.4.2.js`. Information on downloading a (possible themed) version of jQuery UI can be found on `http://jqueryui.com/download`.

Using jQuery UI's themeroller

The theme used throughout this book is called *smoothness* and can be downloaded from `http://jqueryui.com/themeroller/` by choosing the **Gallery** tab and clicking the **Download** button below the **Smoothness** example. It is even possible to create a completely customized theme based on one of the standard themes by selecting one of the themes from the gallery and then tweaking it in the **Roll Your Own** tab. Once you're satisfied with the look you can download the result. Check the online documentation at `http://jqueryui.com/docs/Getting_Started` for all details.

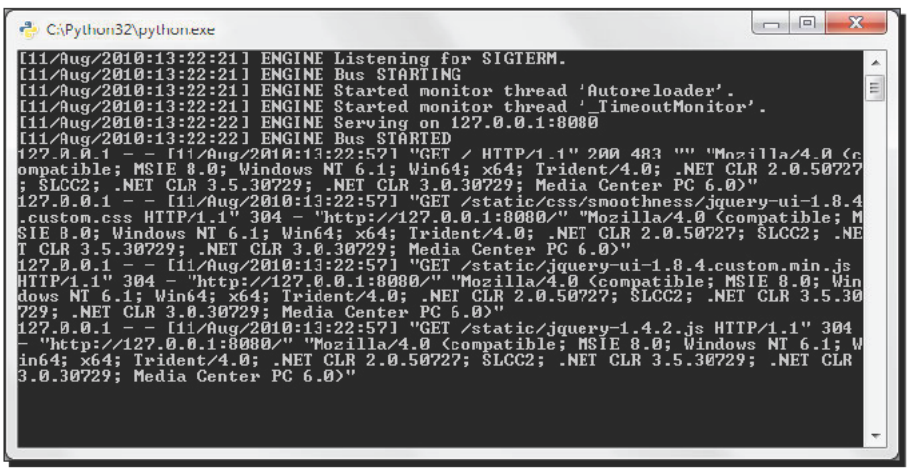
Serving an application

The first task we set ourselves is serving content to the end user. In the end this should be useful content, of course, but let us first make certain that we can write a tiny web application that delivers some content at all.

Time for action – serving a dummy application

Now that we have the necessary building blocks in place, we can start developing our application. Let's start with a very simple application:

1. Go to the directory where you unpacked the example code.
2. Go to the directory `Chapter 2`.
3. Double-click the file `nocontent.py`, a text window will open (alternatively you can enter the command `python nocontent.py` from the command line):



4. Open your favorite browser and enter `http://localhost:8080` in the address bar. You will be presented with a rather dull page:

If your browser is unable to connect to `http://localhost:8080`, this might be because your local name server is not configured to resolve the name `localhost`. If you do not have the means to correct this, it is equally valid, though less convenient, to enter `http://127.0.0.1:8080` in the address bar of your browser.

It is also possible that the default port that the application will be listening on (`8080`) is already in use, in which case, Python will raise an exception: **IOError: Port 8080 not free on '127.0.0.1'**. If that is the case, we can configure CherryPy to listen on a different port (see the info box in the next section).

What just happened?

Double-clicking `nocontent.py` caused the Python interpreter to start and execute the script. This opened up a console window where the CherryPy framework logged the fact that it started and that it will be listening on port `8080` at `127.0.0.1` (the so called loop back IP-address of the local machine, an address present on the machine even if it is not connected to the Internet).

This address and port are the ones we point our browser to, after which the HTTP server provides us with an HTML file, and a couple of JavaScript files to serve the application. Each file that is retrieved by the browser is logged in the console window together with a status. This will be convenient for spotting the missing files, for example.

Our script can be stopped from serving requests by closing the console window or by pressing *Ctrl + Break* (on Windows) or *Ctrl + C* (on Windows and most other platforms).

Time for action – serving HTML as dynamic content

We have seen how to run an application and access it with a web browser, now let's have a look at the Python code needed to accomplish this. We will need to serve static files but in addition to those static files we want to generate the main HTML content dynamically. This isn't strictly necessary as we could have served it as a static file just as easily but it serves as a simple example of how to generate dynamic content:

Chapter2/nocontent.py

```
import cherrypy
import os.path
current_dir = os.path.dirname(os.path.abspath(__file__))
```

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

`nocontent.py` starts off with importing the `cherrypy` and `os.path` modules. The latter is needed to determine the directory that `nocontent.py` resides in (highlighted), so that we may refer to other static files and directories relative to `nocontent.py`. This way, we make life a lot easier once we want to move this application to its final destination on a production server.

Chapter2/nocontent.py

```
class Root(object): ... <omitted> ...

if __name__ == "__main__":
    cherrypy.quickstart(Root(),config={
           '/static':
           { 'tools.staticdir.on':True,
             'tools.staticdir.dir':os.path.join(current_dir,"static")
           }
    })
```

What just happened?

The next step is to start the CherryPy server with the `quickstart()` function (highlighted). We pass two arguments: the first one is an object instance of a class that exposes some methods to CherryPy that may deliver dynamic content. We will look at that one in a minute.

The second (named) argument is a dictionary containing a number of configuration items. In this case, we configure just a static directory, but in other situations, additional configuration items may appear here. The URL component `/static` is made to refer to a location on the file-system relative to `nocontent.py` by concatenating to the `current_dir` determined earlier. Again we use a function from Python's `os.path` module, `os.path.join()`, to create a file path in a platform-independent manner.

The `static` directory contains all jQuery and jQuery UI files we will need for this application along with all CSS files and images to style the application. In this example, without real content there are no additional files besides the ones belonging to the jQuery and jQuery UI libraries, but if we needed them, we could have placed them here.

If we would like CherryPy to listen on a different port, we should indicate this in the global configuration. This can be done by preceding the call to `cherrypy.quickstart()` with `cherrypy.config.update({'server.socket_port':8088})`. CherryPy has a rich palette of configuration options and can even be instructed to read its configuration from files. A good starting point for all the possibilities is `http://www.cherrypy.org/wiki/ConfigAPI`.

We still have to implement a `Root` class to provide CherryPy with an object instance that may act as the root of the document hierarchy that CherryPy may serve. This class should actually be defined before we can create an instance to pass to the `quickstart()` method, but I wanted to concentrate on how to start the server first before concentrating on producing content:

Chapter2/nocontent.py

```
class Root(object):
    content = '''... <omitted> ...'''
    @cherrypy.expose
    def index(self):
        return Root.content
```

This `Root` class contains a single class variable `content` that holds the HTML code we will serve. We will examine it in detail in the next section. This HTML is generated by the `index()` method and passed to the HTTP server that in its turn will pass it on to the requesting browser.

It is **exposed** to CherryPy by the `@cherrypy.expose` decorator (highlighted). Only exposed methods will be called by CherryPy to produce content. In the default configuration, CherryPy will map a URL of the form `/name` to a method called `name()`. A URL containing just a forward slash `/` will map to a method called `index()`, just like the one we defined here. This means we have now configured CherryPy to deliver dynamic content when the user directs his browser to `http://127.0.0.1:8080/` (and he may even omit the final slash as CherryPy effectively ignores a trailing slash by default).

Note that we let `index()` return the contents of a single string variable but we could have returned just about anything, making this a truly dynamic way of producing content.

Who serves what: an overview

Serving an application from a mixture of dynamic and static content may quickly become confusing. It might help to form a clear picture early on of the relations between components, of data streams, and directory structures used. This builds on the general picture sketched in *Chapter 1* and will get extended in each chapter.

Almost all applications in this book are served from the same directory structure:

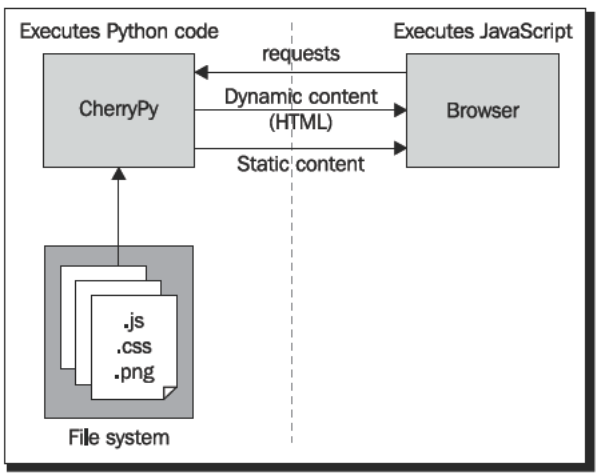
```
├── application_1.py
├── application_2.py
├── module_a.py
├── module_b.py
└── static
    ├── jeditable.js
    ├── jquery-1.4.2.js
    ├── jquery-ui-1.8.4.custom.min.js
    └── css
        └── smoothness
            ├── jquery-ui-1.8.4.custom.css
            └── images
                ├── ui-bg_flat_0_aaaaaa_40x100.png
                └── ui-bg_flat_75_ffffff_40x100.png
```

The top-level directory contains one or more Python files that you can execute and that will start a CherryPy server. Those Python files implement the server-side of an application. They may import additional modules from the same top-level directory.

The top-level directory also contains a subdirectory called `static`. It holds several JavaScript files, including the jQuery and jQuery UI libraries and any additional plugins. It also contains a directory called `css` that contains one or more subdirectories with additional CSS stylesheets and images for jQuery UI themes.

Note that although our applications are served by a web server, there are no HTML files to be seen because all HTML content is generated dynamically.

From an application point of view, the best way to comprehend a web application is to see the application as distributed. Some of its code (in our case Python) runs on the server, while other code (JavaScript) runs in the browser. Together, they make up the complete application as visualized in the following image:



Pop quiz – serving content with CherryPy

We made the choice to serve our content from the `index()` method so users could get the content by referring to the URL ending in just a slash (/). But what if we would like our content to be accessed by referring to a URL like `http://127.0.0.1/content`? What would have to change?

HTML: separating form and content

Almost always, it is a good idea to separate form and content. This enables us to concentrate on the logical structure of the information we want to present and makes it easier to change the appearance of the data later. This even allows for applying themes in a maintainable way.

The structure of our data is laid down in the HTML we deliver to the browser. To be more precise, the structural data can be found within the `<body>` element, but the `<head>` element of the HTML contains important information as well. For example, references to stylesheets and JavaScript libraries that will be used to style the appearance of the data and enhance the user interaction.

In the following code, we use a `<link>` element to refer to a CSS stylesheet from a theme we downloaded from the jQuery UI website (highlighted). In this example, we do not actually use this stylesheet and nor are the jQuery and jQuery UI libraries included in the `<script>` elements, but this example shows how to refer to those libraries from the HTML we produce, and in the following examples, we will see that this is also the spot where we refer to any additional JavaScript libraries that we will create ourselves. The actual content is enclosed in the highlighted `<div>` element.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
<link rel="stylesheet"
 href="static/css/redmond/jquery-ui-1.8.1.custom.css"
 type="text/css" media="screen, projection" />
<script type="text/javascript"
   src="static/jquery-1.4.2.js" ></script>
<script type="text/javascript"
   src="static/jquery-ui-1.8.1.custom.min.js" ></script>
</head>
<body id="spreadsheet_example">
<div id="example">an empty div</div>
</body>
</html>
```

Time for action – a unit convertor

Serving just a piece of text isn't very useful, so our next step is to add some HTML content and enhance the display and functionality with JavaScript:

1. Go to the same directory where `nocontent.py` could be found.
2. Double-click the file `unitconvertor.py`, CherryPy console will again open in a text window.
3. Enter `http://localhost:8080` in the address bar of your browser (or click refresh if it is still open on that address). You will now see a small unit convertor:

You can enter any number (with an optional fraction) in the text input on the left and after selecting the units to convert from and to, pressing the **convert** button will present you with the converted number on the right.

What just happened?

The basic structure of our web application hasn't changed. The content we deliver is different but that hardly changes the Python code we need to deliver it. The actual content, that is the HTML we deliver when the `index()` function is invoked, does differ as it has to define the `<form>` elements that our unit convertor consists of and we want to execute some JavaScript as well.

HTML: form-based interaction

The `<head>` portion of the HTML doesn't have to be changed as it already refers to the stylesheet and JavaScript libraries we want to use. However, we do have to change the `<body>` element to contain the structural elements that make up our unit convertor.

The unit convertor is structured as a `<form>` element (highlighted). It contains two drop-down lists to select the units to convert, both implemented with `<select>` elements, and a text `<input>` element where the user can enter a number. A second text `<input>` element is used to display the result of the conversion. This one is set to read only as it is not meant to receive input from the user. The final element is a `<button>` that the user may click to initiate the conversion.

You may have noticed that the `<form>` element lacks an `action` attribute. This is intentional as there is no interaction with a server. The conversion that happens when the user clicks the button is completely implemented in JavaScript. This JavaScript is included (and executed) in the final script element (highlighted). We will examine this script in the next section.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
<link rel="stylesheet" href="static/css/redmond/jquery-ui-
1.8.1.custom.css" type="text/css" media="screen, projection" />
<script type="text/javascript" src="static/jquery-1.4.2.js" ></script>
<script type="text/javascript" src="static/jquery-ui-1.8.1.custom.min.
js" ></script>
</head>
```

```
<body id="spreadsheet_example">
<div id="example">
    <form id="unitconversion">
    <input name="from" type="text" value="1" />
    <select name="fromunit">
           <option selected="true">inch</option>
           <option>cm</option>
    </select>
    <label for="to">=</label>
    <input name="to" type="text" readonly="true" />
    <select name="tounit">
           <option>inch</option>
           <option selected="true">cm</option>
    </select>
    <button name="convert" type="button">convert</button>
    </form>
</div>
<script type="text/javascript" src="unitconverter.js" ></script>
</body>
</html>
```

JavaScript: using jQuery UI widgets

Screen elements or **widgets** are essential to let the end user interact with you application. These widgets might be simple buttons that initiate some action when the user clicks them or more complex widgets like drop-down boxes, radio buttons, or even little calendars that let you pick a date. The jQuery UI library provides a large number of predefined and easy to configure widgets, and so our next step is to use jQuery UI to let the button in our conversion application react to a click of the mouse and initiate the unit conversion.

Time for action – conversion using unitconverter.js

`unitconverter.js` contains the necessary JavaScript code to do the actual conversion. It starts with the definition of a conversion map, a dictionary holding the conversion factors for any conversion we want to define. We restrict ourselves to conversions from inches to centimeters and vice versa, but additional conversion factors can easily be added.

```
conversion_map = {
    "inch cm":1.0/2.54,
    "cm inch":2.54
};
$("button").button().click(function(){
           value=$("input[name='from']").val();
           f=$("select[name='tounit'] option:selected").val();
```

```
            t=$("select[name='fromunit'] option:selected").val();
            if(f != t){
                  c=conversion_map[f+' '+t];
                  result=parseFloat(value)*c;
            }else{
                  result = value;
            }
            $("input[name='to']").val(result);
      }
);
$("form *").addClass("ui-widget");
```

The highlighted line in the previous code is our first encounter with the jQuery and jQuery UI libraries and deserves some close attention. The `$("button")` part selects all `<button>` elements on the page. In this case, it will be just a single one. This `<button>` element is converted to a button widget from the jQuery UI library with the `button()` method. This is a simple widget that styles an element as a recognizable button that will be easy to theme and customize.

What just happened?

What actually happens once the user clicks the button is defined by the anonymous function we pass as a **click handler** to the button element with the `click()` method. This anonymous function is called each time the user clicks the button.

The first thing this handler does is retrieve the contents of the text `<input>` element with a `name` attribute equal to **from** with `$("input[name='from']").val()`. Next, it retrieves the currently selected units from both `<select>` elements.

If those units are not the same, it fetches the conversion factor from the conversion map with the concatenated units as a key. The result of the conversion is calculated by multiplying the conversion factor and the contents of the `<input>` element. The content we retrieve of any `<input>` element is always returned as a string, therefore we have to use the built-in JavaScript function `parseFloat()` to interpret it as a floating point number. If both units are equal, the result is simply the same as the input value.

The calculated result is stored in the text `<input>` element with a `name` attribute of `to`. Note that even though this element has a read-only attribute to prevent the user from entering any text, we can still alter its content within a script.

Pop quiz – adding an icon to a button

A button with just simple text might be appropriate for many applications but it would look much better, if it showed an appropriate icon as well. Knowing that the button widget is highly configurable, how would you add an icon to your button?

Have a go hero – adding a dynamic title

- The HTML we served in the `nocontent.py` example was simply the contents of a class variable, so not really dynamic! What all would we have to do if we wanted to serve HTML containing a `<title>` element that shows the current date?
- Hint: A `<title>` element should be contained inside the `<head>` element. So instead of returning all the HTML in one go, you could rewrite the Python code to return HTML composed of three parts: The first and last parts are pieces of static HTML and the middle part is a dynamically generated string representing a `<title>` element containing a date. That date could be obtained from Python's `asctime()` function found in the standard `time` module.
- A possible implementation can be found in the file `nocontenttitle.py`.

jQuery selectors

jQuery selectors pop up in many locations and in a sense they are the focus of any JavaScript program that uses the jQuery library. A complete overview is out of the scope of this book (for that, refer to the appendix for some books on jQuery with extensive examples or check jQuery's documentation section on `http://docs.jquery.com/Main_Page`, especially the part on selectors) but basically jQuery allows us to select any element or set of elements in a CSS 3-compliant fashion in a cross browser compatible way. In other words, it works even in browsers that do not yet support CSS 3.

To give some idea of what is possible, some examples are given next, all of them assume an HTML document containing the following markup:

```
<div id="main">
<ul>
<li>one</li>
<li class="highlight">two</li>
<li>three</li>
</ul>
</div>
<div id="footer">footer text</div>
```

- To select all `<li>` elements: `$("li")`
- To select just the first `<li>` element: `$("li:first")`
- To select the `<li>` element with the class `highlight`: `$(".highlight")`
- To select the `<div>` with an ID equal to `footer`: `$("#footer")`

The jQuery function (often represented by the alias `$`) returns a jQuery object that refers to the collection of matched elements. A jQuery object has many methods to manipulate the elements in this collection. For example, `$("li").addClass("red-background")` adds the red-background class to all `<li>` elements.

The jQuery UI library extends the available methods even further by adding functionality to change elements to standardized widgets. That is why in our example `$("button").button()` alters the appearance of our button element to the stylized button widget that jQuery UI provides.

Our example application also shows another important jQuery concept: **chaining**. Most jQuery and jQuery UI methods return the selection they operated on. That way, it is easy to call multiple methods on the same selection. In our example, `$("button").button()` returns the selected button elements after transforming them into button widgets, which allows us to chain the click method to define mouse-click behavior by writing `$("button").button().click(…)`.

CSS: applying a jQuery UI theme to other elements

The last line in `unitconverter.js` shows how we can style any element in the same manner as the standard jQuery UI widgets. This is accomplished, in this case, by selecting all elements contained in the `<form>` element with `$("form *")` and then adding the `ui-widget` class with the `addClass()` method.

Any element adorned with the `ui-widget` class will receive the same styling as any jQuery UI widget. In our case, this is visible in the font and colors used in the `input` and `select` elements. Even if we change the theme this change will be uniformly applied. There are more predefined classes available to allow for a more fine grained control and we will encounter those when we create our own jQuery UI plugin in the next section.

It is important to grasp the effect of one of the predefined jQuery UI classes to an element. Classes in themselves don't change the way elements are displayed but the jQuery UI framework associates various CSS style elements with the predefined classes. When the classes associated with an element change, the browser checks again which style elements to apply, effecting an immediate style change.

It is also possible to directly alter CSS styles associated with an element. However, defining styles for a certain class and altering the class makes it easier to maintain a consistent look without having to resort to individual style components for each and every element that you want to change.

Have a go hero – adding zebra stripes to a table

An often required feature when styling HTML tables is to render the rows of tables with an alternating background color.

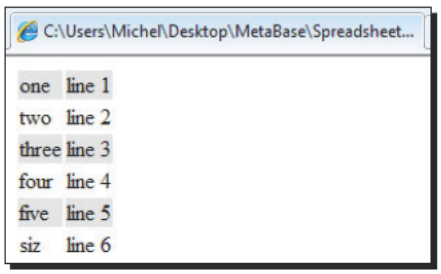
Because jQuery allows us to use CSS 3-compliant selectors and add to an elements `class` attribute with the `.addClass()` method, this is now accomplished easily even in the browsers that do not support CSS 3.

Given the following sample HTML, what JavaScript should be added to the last `<script>` element to render the background of all even rows in light gray? (Hints: CSS 3 has an `:even` selector and when you add a class to an element with jQuery, any CSS styles applicable to that class are re-evaluated).

Check `zebra.html` to see a solution (It is included with the sample code for *Chapter 2*. Open the file in your browser to see the effect):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
  <script type="text/javascript" src="static/jquery-1.4.2.js"
  ></script>
  <style>
    .light-grey { background-color: #e0e0e0; }
  </style>
</head>
<body>
  <table>
    <tr><td>one</td><td>line 1</td></tr>
    <tr><td>two</td><td>line 2</td></tr>
    <tr><td>three</td><td>line 3</td></tr>
    <tr><td>four</td><td>line 4</td></tr>
    <tr><td>five</td><td>line 5</td></tr>
    <tr><td>siz</td><td>line 6</td></tr>
  </table>
  <script type="text/javascript">
  /* insert some JavaScript here to color even rows grey */
  </script>
</body>
</html>
```

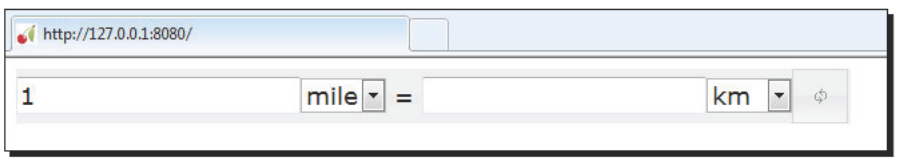
The result will look something like this in the browser (note that elements are numbered starting at zero, so maybe the result is not what you expected):



Time for action – converting a unit convertor into a plugin

Re-using one of the many well designed jQuery UI widgets is good as it saves us development and maintenance time but the true power of the jQuery UI framework is the manner in which it enables us to devise completely new widgets that merge seamlessly with the rest of the framework and are indistinguishable in their use from the standard widgets. To illustrate what is possible, let's implement our unit converter again, but this time as a jQuery plugin:

1. Go to the directory containing the example code for *Chapter 2*.
2. Double-click the file `unitconverter2.py`, the CherryPy console will again open in a window.
3. Enter `http://localhost:8080` in the address bar of your browser (or click refresh if it is still open on that address). You will now see a slightly restyled unit converter:



The interaction with this new unit converter is exactly the same as our previous one.

What just happened?

Instead of structuring a widget with a `<form>` element containing a number of additional elements, we now take a simpler approach. We will design a reusable unit converter widget that can be inserted into any `<div>` element. Our HTML backbone becomes much simpler now, as its body will just contain a single `<div>` element:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
```

```
<link rel="stylesheet" href="static/css/redmond/jquery-ui-
1.8.1.custom.css" type="text/css" media="screen, projection" />
<script type="text/javascript" src="static/jquery-1.4.2.js" ></script>
<script type="text/javascript" src="static/jquery-ui-1.8.1.custom.min.
js" ></script>
<script type="text/javascript" src="unitconverter2.js" ></script>
</head>
<body id="spreadsheet_example">
<div id="example"></div>
<script type="text/javascript">
$("#example").unitconverter(
{
'km_mile':1.0/0.621371192,
'mile_km':0.621371192
});
</script>
</body>
</html>
```

The first highlighted line includes the JavaScript file that contains the new implementation of the unit converter. We refer to the plugin defined in this file in the JavaScript code near the end of the `<body>` element (last highlighted line). This script refers to the `<div>` element to which we want to add a unit converter by its ID (in this case `#example`) and apply the `unitconvertor()` method.

As we will see when we look at the JavaScript code that implements our converter plugin, `unitconverter()` takes an option object as its single argument. This option object may contain any number of keys defining additional conversion factors for this instance of the plugin. In this case, we pass additional information to allow for conversion from miles to kilometers, and vice versa.

Pop quiz – adding conversions to a unitconverter instance

What would the JavaScript look like when we want to add a unit converter plugin with the possibility of converting from cubic feet to liters?

JavaScript: creating a jQuery UI plugin

All jQuery UI plugins are defined in the same way by adding a new function to the `fn` attribute of the `jQuery` object (the object we mostly refer to by its alias `$`). In `unitconverter2.js`, this is exactly what we do, as it is seen in the first line of the following code.

The next thing we do is merge any options passed to the plugin with defaults (highlighted). jQuery provides an `extend()` method that merges the attributes of any number of objects and returns the first one. As we do not want to overwrite the default options that we have defined in `$.fn.unitconverter.conversion_map`, we pass it an empty object. This object will receive the default attributes and any attributes defined in the `options` object, overwriting the ones with a name that is the same. This set of merged attributes is stored in the `cmap` variable:

```
jQuery.fn.unitconverter = function(options){
    var cmap = $.extend({},$.fn.unitconverter.conversion_map,options);
```

The conversion factors are referred to by keys of the form `unit1_unit2`. To construct two drop-down selectors from the keys, we iterate over all these keys and use JavaScript's `split()` method to retrieve the individual units (highlighted). These are then stored in the `from` and `to` arrays:

```
    var from = new Array();
    var to = new Array();
    for (var key in cmap){
        var units = key.split("_");
        from.push(units[0]);
        to.push(units[1]);
    }
```

The next step is to construct the HTML needed by the plugin to present to the user. The structure is similar to the handcrafted one used in the previous example, a `<form>` with `<input>` and `<select>` elements, and a `<button>`. The `<form>` element is adorned with a random ID attribute. This way we may refer to it later even if there is more than one unit converter present on the page.

The `<select>` elements contain a number of `<option>` elements that are created by retrieving the unit names stored in the `from` and `to` arrays one-by-one with the `pop()` method. The first of these options is selected by default (highlighted). The HTML code is then passed to the `append()` method of `this`. `this` is a variable that is available to the function implementing the plugin that contains the selected elements the plugin is applied to, in our example the `<div>` element with the `#example` ID:

```
    var id = "unitconverter" + new String(Math.floor(Math.random() *
255 * 255));
    var html = '<form id="' + id + '"><input name="from" type="text"
value="1" />';
    html += '<select name="fromunit">';
    html +=      '<option selected="true">'+from.pop()+'</option>';
    var len = from.length;
    for (var i=0; i<len; i++){
```

```
    html += '<option>' + from.pop() + '</option>' };
        html += '</select> = ';
        html += '<input name="to" type="text" readonly="true" />';
        html += '<select name="tounit">';
        html += '<option selected="true">' + to.pop() + '</option>';
        var len = to.length;
        for (var i=0; i<len; i++){
    html += '<option>' + to.pop() + '</option>'};
        html += '</select>';
        html += '<button name="convert" type="button">convert</button>'
    html += '</form>';
        this.append(html);
```

The randomly generated ID for the form element now comes in handy to select just the `<button>` element within the form we are currently constructing and convert it to a button: we construct a suitable selector by concatenating relevant parts with `"#"+id+" button"`.

Note that it is perfectly valid to include other plugins or widgets within a custom plugin. This time we choose to construct a slightly different looking button with just an icon and no text by passing an appropriate options object. From the numerous icons shipped with jQuery UI, we choose the one that represents the function of the button best: `ui-icon-refresh` (highlighted).

The conversion that happens when the user clicks the button is implemented by a function that we will encounter shortly and that is passed by the button object (available to the `click()` method as the `this` variable) and the merged map of conversion factors:

```
    $("#"+id+" button").button({
                icons: {
                        primary: 'ui-icon-refresh'
                },
                text: false
        }).click(function(){return convert(this,cmap);});
```

The finishing touch is to style our widget in a consistent manner. jQuery provides us with a `css()` method that allows us to directly manipulate the style attributes of any element. We first deal with a layout matter: we apply a `float:left` style to the `<form>` element to make sure it doesn't fill the page completely, but shrink/wraps itself around the elements it contains:

```
    $("#"+id).css('float','left');
```

We then copy a number of background style attributes from the `<button>` element to the `<form>` element to give the `<form>` element a look that is consistent with the theme applied to the standard button widget. Other style elements from the theme like font face and font size are applied to the form element by adding the `ui-widget` class (highlighted). We end by returning the `this` variable (which in our example contains the `<div>` element we selected, but now with the `<form>` element we just added to it). This allows for chaining additional jQuery methods:

```
    $("#"+id).css('background-color',
$("#"+id+" button").css('background-color'));
    $("#"+id).css('background-image',
$("#"+id+" button").css('background-image'));
    $("#"+id).css('background-repeat',
$("#"+id+" button").css('background-repeat'));
    $("#"+id).addClass("ui-widget");
    return this;
};
```

Of course, we still need to define a function that does the actual conversion when the button of the unit converter is clicked. It differs slightly from the previous implementation.

The `convert()` function is passed both the `<button>` element that is clicked and a map with conversion factors. The `<form>` element enclosing the button is determined with the `parent()` method and stored in the `form` variable.

The input value we want to convert is retrieved from the `<input>` element with a `name` attribute equal to `from`. We can find this specific element by selecting all children of the `<form>` element stored in `form` and filtering these children by passing a suitable selector to the `.children()` method (highlighted).

In a similar way, we determine which option is selected in the two `<select>` elements:

```
function convert(button,cmap){
    var form = $(button).parent();
    var value = form.children("input[name='from']").val();
    var f = form.children("select[name='tounit']").
children("option:selected").val();
    var t = form.children("select[name='fromunit']").
children("option:selected").val();
```

What is left is the actual conversion. If the conversion units are not equal, we retrieve the conversion factor from the map (highlighted) and then multiply it by the contents of the `<input>` element interpreted as a floating point number. If the input can't be interpreted as a floating point number or there wasn't a suitable conversion factor in the map, the result of the multiplication is a `NaN` (Not a Number) and we signal that fact by placing an error text in the result. However, we convert the result to a number with four decimal digits with JavaScript's `toFixed()` method if everything goes well:

```
var result = value;
    if(f != t){
           var c=cmap[f+'_'+t];
           result=parseFloat(value)*c;
           if (isNaN(result)){
                  result = "unknown conversion factor";
           }else{
                  result = result.toFixed(4);
           }
    }
    form.children("input[name='to']").val(result);
};
```

`unitconverter2.py` concludes by defining an object with defaults:

```
jQuery.fn.unitconverter.conversion_map = {
    "inch_cm":1.0/2.54,
    "cm_inch":2.54
}
```

Pop quiz – changing option defaults

If we would:

1. Add a unitconvertor to a element with an ID #first.
2. Add the possibility of converting from cubic feet to liters to the default conversion map.
3. And finally, add a unitconverter to a element with an id #last.

The code would look something like this:

```
$("#first").unitconverter();
$.extend($.fn.unitconverter.conversion_map, {'cubic feet_
litres':1.0/28.3168466});
$("#last").unitconverter();
```

If we would execute the preceding code, which element(s) would get a unitconverter with the added conversion possibility?

a. The div with the #first ID

b. The div with the #last ID

c. Both

Designing a spreadsheet application

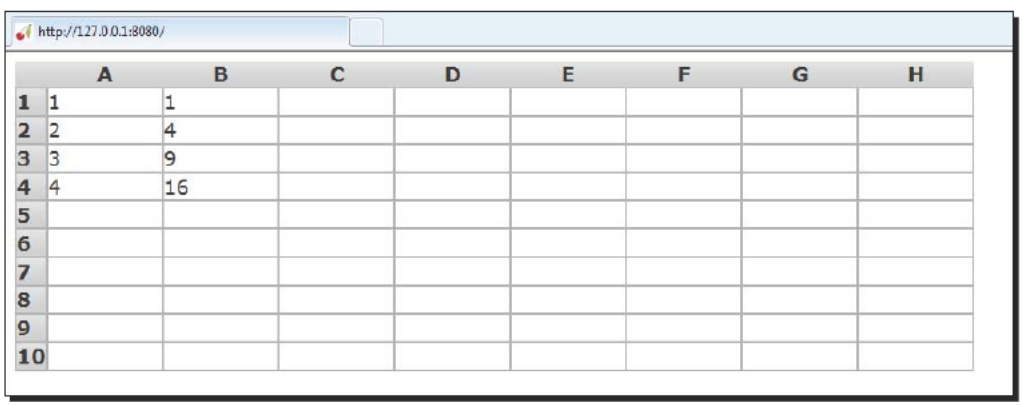
Our goal for this chapter was to be able to present the user with a simple spreadsheet application and we are nearly there. We know how to serve HTML and we saw how we can implement a custom jQuery UI widget, so let's apply that knowledge to designing a spreadsheet plugin. First let's see how it will look:

Time for action – serving a spreadsheet application

Go to the directory containing the example code for *Chapter 2*:

1. Double-click the file `spreadsheet.py`, the now familiar CherryPy console will open in a text window.

2. Enter `http://localhost:8080` in the address bar of your browser (or click refresh if it is still open on that address). You will now see a simple spreadsheet application:



	A	B	C	D	E	F	G	H
1	1	1						
2	2	4						
3	3	9						
4	4	16						
5								
6								
7								
8								
9								
10								

3. You can click on any cell to edit its formula. You should not start a formula with an equal sign: **42**, **D2+19** and **"text"** (including the double quote marks) are examples of valid formulas. In fact, any JavaScript expression is valid.

What just happened?

The spreadsheet application served to the end user consists of two major parts, HTML to structure the spreadsheet and some JavaScript to provide interaction. We look at each of these in turn.

HTML: keeping it simple

The HTML we need for our spreadsheet is nearly identical to the one for the unit converter. The highlighted lines in the following code show the differences. `spreadsheet.js` contains the definition of the plugin and the final `<script>` element inserts an 8x10 spreadsheet into the `#example` div. Converting a `<div>` element to a fully functional spreadsheet widget is just as simple as converting to the standard button widget!

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
<link rel="stylesheet"
href="static/css/redmond/jquery-ui-1.8.1.custom.css" type="text/css"
media="screen, projection" />
<script type="text/javascript"
 src="static/jquery-1.4.2.js" ></script>
<script type="text/javascript"
 src="static/jquery-ui-1.8.1.custom.min.js" ></script>
<script type="text/javascript"
 src="static/jeditable.js" ></script>
<script type="text/javascript"
 src="spreadsheet.js" ></script>
</head>
<body id="spreadsheet_example">
<div id="example"></div>
<script type="text/javascript">
$("#example").sheet({cols:8,rows:10});
</script>
</body>
</html>
```

JavaScript: creating a spreadsheet plugin

The file `spreadsheet.js` contains all JavaScript code needed to implement a reusable spreadsheet widget. The spreadsheet is very similar to our unit converter from a jQuery perspective although the actual JavaScript to implement the user interaction is somewhat more involved. Again our plugin is a function that is associated with jQuery's `fn` attribute as it can be seen in the very first line in the following code, where we define our widget with the name `sheet`.

Next we merge the default options for the sheet plugin (defined at the end of the file) with the options passed to the function:

```
jQuery.fn.sheet = function(options){
   var opts = $.extend({}, $.fn.sheet.defaults, options);
```

The next step is to create a table that will represent our spreadsheet. We create this `<table>` element with a number of associated classes: its very own `sheet` class to make it easily recognizable as a sheet plugin once created, a `ui-helper-reset` class that will cause suitable CSS to be applied by jQuery to reset any unwanted default styling added by the browser and finally a `ui-widget` class that will cause the selected theme to be applied. Then we create the table contents step-by-step by adding the needed HTML to a variable `t` in incremental steps:

```
   /* create a cols x rows grid */
   var t='<table class="sheet ui-helper-reset ui-widget"
    cellspacing="0">';
```

The table contains a `<thead>` element that will be styled as a `ui-widget-header`. It contains a single row of `<th>` elements. These `<th>` elements contain the column label, a capital letter that we construct from the column index with the `fromCharCode()` method (highlighted):

```
   t=t+'<thead class="ui-widget-header">
 <tr class="ui-helper-reset"><th></th>';
   for(i=0;i<opts.cols;i=i+1){
   t=t+'<th class="ui-helper-reset">' +
String.fromCharCode(65+i)+"</th>";
   }
```

The body of the table consists of a `<tbody>` element containing a number of rows with `<td>` elements. The first `<td>` element of each row contains the row label (a number) and will be styled as a `ui-widget-header` just like the column labels. The regular cells, that is the ones that will contain our formulas and values, will belong to the class `ui-widget-content` to style them in an appropriate manner. These cells will also belong to a class `cell` to make them easy to distinguish when we add additional functionality to them (highlighted).

There is initially no content in such a cell except for a `<span>` element that will contain the formula and that will be styled as `ui-helper-hidden`, rendering the formula invisible. The value of the evaluated formula will be stored both as text content in the `<td>` element (side-by-side with the `<span>` element) and as a global variable with a name equal to the name of the cell. A global variable in this context is a named attribute of the top-level `window` object defined by the browser that may be accessed as `window[name]`.

Storing the value of a cell in a global variable as well allows us to use any JavaScript expression as the formula in a cell because we can now refer to the value of any other cell by name. `A1+B3*9`, for example will be a perfectly valid expression because `A1` and `B3` will be defined as global variables:

```
t=t+'</tr></thead><tbody class="ui-widget-content" >';
for(i=0;i<opts.rows;i=i+1){
       t=t+'<tr class="ui-helper-reset">
       <td class="rowindex ui-helper-reset ui-widget-header">'
              + (i+1)+"</td>";
       for(j=0;j<opts.cols;j=j+1){
         id=String.fromCharCode(65+j)+(i+1)
         t=t+'<td class="cell ui-helper-reset ui-widget-content"
         id="'+id+'">
                <span class="formula ui-helper-hidden">
                </span></td>';
              /* create a global variable */
              window[id]=0
       }
       t=t+"</tr>";
}
t=t+"</tbody></table>";
this.append(t);
```

The HTML for the table we created in the `t` variable is then inserted into the jQuery selection that we applied the `sheet()` method with the `.append()` method of the `this` object. The `this` object is available to any function defining a plugin and holds the current jQuery selection.

To edit a cell, we will employ the jEditable plugin. This plugin will take care of the user interaction when the user clicks a cell to edit its content. To do this it needs functions to get and set the contents of a cell.

The jEditable plugin we use here is included in the example code distributed with this chapter. The latest version can be obtained from Mika Tuupola's website: `http://www.appelsiini.net/projects/jeditable`. It comes with a pretty comprehensive set of documentation. Turning a `<td>` element into something that changes into an editable textbox when the user clicks on it with a mouse, is as simple as selecting the element and invoking the `editable()` method. For example, `$(".editable").editable("http://www.example.com/save")` will render any element with the `editable` class into an editable textbox once clicked and will send the edited contents to the URL passed as the first parameter to the `editable()` method. The jEditable plugin comes with a host of options and we will encounter a few of them when we employ the jEditable plugin to do the editing of the spreadsheet cells.

We need to define a function that will be invoked by jEditable for extracting the content of the element. This function will require two arguments:

1. The element we are editing (a `<td>` element in our example).
2. The original settings passed to the jEditable plugin. Those settings we ignore for now.

The `<td>` elements are structured in such a way that the formula itself is stored in a (hidden) span element. The `getvalue()` function then must get access to this `<span>` element first before it can obtain the formula.

Therefore, we convert the `<td>` element first to a jQuery object (highlighted) and then filter the elements it contains to just elements with a class of `formula`. This amounts to just the `<span>` element whose text is the formula we are after:

```
function getvalue(org, settings){
        d=$(org)
        return d.filter(".formula").text()
}
```

The corresponding `setvalue()` function is used by jEditable to store the edited formula again in the `<td>` element. When called this function is passed two arguments:

1. The edited content of the element.
2. The original settings passed to the jEditable plugin and its code is quite complicated because storing the formula is not the only thing it has to do. It must also calculate the result of the formula and update any cells that depend on the updated cell.

The cell we are editing (that is the `<td>` element) is available as the `this` variable. We stored the cell index as its `id` attribute so we retrieve that one first (highlighted). The `value` argument that was passed to the `setvalue()` function is the edited formula.

As we use JavaScript syntax for these formulas, we can simply call JavaScript's `eval()` function to calculate the value of the formula. We have to store the result in global variables with the name of the cell as well to make it reusable by other cells. Note that these global variables are just attributes of the `window` object in the context of the browser so assigning a value to such an attribute is just what we do inside the `if ... else ...` clause. If the result of evaluating the formula was undefined in some way (for example, because of an error) we set the result to the string `'#undef'` to signal that situation to the user:

```
function setvalue(value, settings) {
        /* determine cell index and update global var */
        currentcell=$(this).attr( 'id');
        currentresult=eval(value);
        if (typeof(currentresult) == 'undefined'){
                currentresult='#undef';
```

```
        window[currentcell]=0;
}else{
        window[currentcell]=currentresult;
}
```

After we have evaluated the formula of the current cell and stored its result we must now recalculate all other cells because they may depend on the contents of the cell we just changed.

We do affect this by selecting all cells in the sheet and applying a function to each of them (highlighted). If we are looking at a different cell than the one just changed (something we determine by comparing their `id` attributes), we recalculate the formula contained in its `<span>` element. If the result is different from the previous value stored for a cell we set the change variable to true. We repeat the whole procedure until nothing is changed or we repeated ourselves more often than there are cells in the sheet, at which point we must have a circular reference somewhere, something we indicate to the user by setting the value of the cell to a suitable text. This is certainly not the most efficient method to recalculate a spreadsheet, nor is it a failsafe method to detect all circular references but it works well enough:

```
/* update all other cells */
var changed;
var depth = 0;
do{
        depth++;
        changed = false;
        $('.sheet').find('.cell').
                each(function (index,element){
                cell=$(element).attr('id');
                if(currentcell != cell){
                        span=$(element).
                                children('span').first();
                        orig = window[cell];
                        window[cell]=0;
                        formula=span.text();
                        if(formula.length > 0){
                                result=eval(formula);
                                if (result != orig) {
                                        changed = true;
                                }
                                if(typeof(result)=='undefined'){
                                        result='#undef';
                                }else{
                                        window[cell]=result;
                                }
                        }else{
                                result = ' ';
                        }
```

```
                                $(element).empty().
    append('<span class="formula ui-helper-hidden replaced">' +
    formula+'</span>'+result);
                            }
                    });
            }while(changed && (depth <opts.cols*opts.rows));
            if ( depth >= opts.cols*opts.rows){
                    currentresult = '#Circular!';
            }
            return('<span
                    class="formula ui-helper-hidden">'
                            +value+'</span>'+currentresult);
        }
```

The purpose of defining functions to set and get a value from a `<td>` element was to be able to apply the jEditable plugin to every cell. This we do in the final lines of our `sheet` plugin. We find all children with a `cell` class (highlighted) and invoke an anonymous function on each of them.

This function first applies the jEditable plugin on the element by invoking the `editable()` method with a reference to our `setvalue()` function as the first argument and an options object as the second argument. The `type` attribute marks this editable element as a text element (and not, for example, a multiline text area element), whereas setting `onblur` to `cancel` indicates that on clicking outside the cell when editing will revert the content to its original. The `data` attribute points to our `getvalue()` function to indicate to the plugin how to get the value that we want to edit.

The second thing the function does is applies CSS style attributes to each cell. In this case a fixed `width` and the `border-collapse` attribute will make sure that the border between cells is just as wide as the border on outlying cells:

```
        /* make every cell editable with the jEditable plugin */
        this.find(".cell").each(function (index,element) {
        $(this).
         editable(setvalue,{type:'text',onblur:'cancel',data:getvalue})
        });
        $(".cell").css({'width':opts.width,'border-collapse':'collapse'});
        return this;
    }
```

`spreadsheet.js` is completed with the definition of a default options object:

```
    jQuery.fn.sheet.defaults = {
        rows : 4,
        cols : 4,
        width: '100px',
        logging: false
    }
```

Have a go hero – adding math functions

In the spreadsheet we designed, the user may use any JavaScript expression as the cell formula. That's fine if we want to use operators like addition (+) or multiplication (*), but what if we would like to use, for example, trigonometric functions like `sin()` or `cos()`?

This is possible by referring to the methods of the built-in JavaScript object `Math` (an example would be `Math.sin(A1)+Math.cos(B1)`) but prefixing every function with `Math` is awkward. Devise a way to make these functions available without the `Math` prefix. (Hint: we already saw how to create names in the global namespace by assigning to `window[<name>]`).

A solution can be found in `spreadsheet2.js`. Its effects can be tested by running `spreadsheet2.py`.

The missing parts

In designing and building a spreadsheet application we saw that it is relatively simple to implement quite sophisticated user interaction by making full use of the jQuery and jQuery UI libraries and choosing wisely from the wide array of available additional plugins like jEditable.

However, although our spreadsheet application was served from the CherryPy server, the functionality of the application was limited to client-side activity only. For example, there is no possibility to save or load a spreadsheet on the server, and neither is there a way to limit the access to our spreadsheet to authorized users only. Both requirements depend on ways to store data in a persistent manner and dealing with persistence will be the next step on our road to developing web applications.

Summary

We have learned a lot in this chapter. Specifically, we covered:

- How to create an environment to develop and deliver our applications. We saw how to install Python, CherryPy, and the jQuery and jQuery UI frameworks.
- The design of a simple spreadsheet application.
- How to configure CherryPy to deliver static and dynamic content.
- How to use standard jQuery UI widgets and third party plugins; specifically, the button widget and the jEditable plugin.
- The implementation of our own jQuery plugin.

We also discussed how to reuse jQuery UI's concept of `ui-widget` classes to style our own widget components in a way that blends seamlessly with jQuery UI's themes.

Now that we've learned about the client-side of web applications, we're ready to tackle server-side issues—which is the topic of the next chapter.

3
Tasklist I: Persistence

In the previous chapter, we learned how to deliver content to the user. This content consisted of HTML markup to structure the information together with a number of JavaScript libraries and code to create a user interface.

We noted that this was not a full-fledged web application yet; it lacked the functionality to store information on the server and there was no way to identify different users or any way to authenticate them. In this chapter, we will address both these issues when we design a simple tasklist application.

This tasklist application will be able to serve multiple users and store the list of tasks for each user on the server.

Specifically, we will look at:

- How to design a tasklist application
- How to implement a logon screen
- What a session is and how this allows us to work with different users at the same time
- How to interact with the server and add or delete tasks
- How to make entering dates attractive and simple with jQuery UI's datapicker widget
- How to style button elements and provide tooltips and inline labels to input elements

Designing a tasklist application

Designing an application should start with a clear idea of what is expected. Not only to determine what is technically required, but almost as important, to define clear boundaries so that we don't lose time on things that are just nice to have. Nice to have features are something to be added if there is time left in the project.

So let's draw up a shortlist of the relevant features of our tasklist application. Some of these may seem obvious, but as we will see, these have a direct impact on some implementation choices that we have to make, such as:

- The application will be used by multiple users
- Task lists should be stored indefinitely
- A task list may contain an unlimited number of tasks but the user interface is designed for optimal performance for up to 25 tasks or so
- Tasks may be added, deleted, and marked as done

Although this list isn't exhaustive, it has some important implications.

The fact that the tasklist application will be used by more than one user means that we have to identify and authorize people who want to use it. In other words, we will need some sort of logon screen and a way to check people against some sort of password database. Because we do not want to burden the user with identifying himself/herself each and every time a task list is refreshed or altered, we need some way of implementing the concept of a **session**.

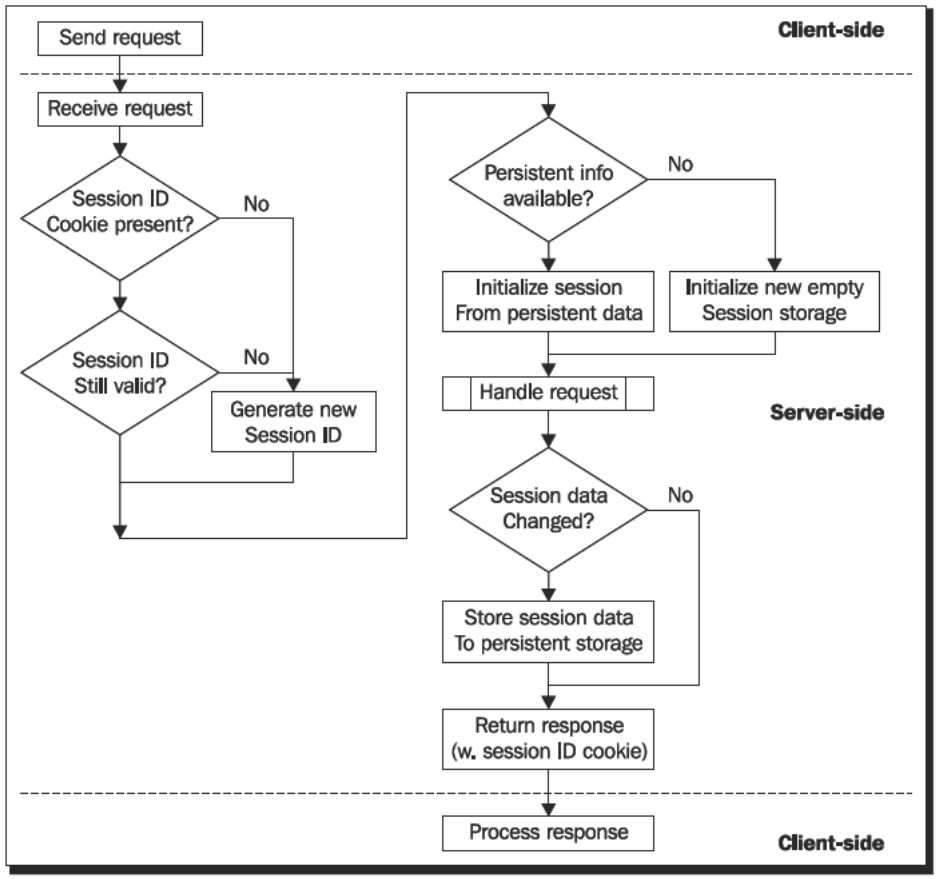
Web applications use the stateless HTTP protocol. This means, from the server's point of view, every request is a single, unrelated event, and no information is retained at the server. This obviously presents us with a problem if we want to perform a set of related actions. The solution is to ask the web browser to send a small piece of information along with every request it makes to the application after the application has identified the user.

This might be accomplished in a number of ways. The server may add an extra parameter to all links inside any web page it generates, commonly referred to as a **session id**, or use the even more general concept of a **cookie**.

Once the server asks the web browser to store a cookie, this cookie is sent with every following request to the same website. The advantage of cookies is that common web application frameworks (like CherryPy) are already equipped to deal with them and implementing sessions with cookies is much simpler than designing the application to alter all hyperlinks it generates to include a proper session ID. The disadvantage might be that people may block their browser from storing cookies because some websites use them to track their clicking behavior.

We let the simplicity of implementation prevail and opt for cookies. If users want to block cookies this is not much of a problem as most browsers also have the option to selectively allow cookies from designated websites.

The following image illustrates the way CherryPy manages sessions with the help of cookies:



It starts when the client (the web browser) sends a request to CherryPy. Upon receiving the request, the first check is to see if the web browser has sent along a cookie with a session ID. If it didn't, a new session idea is generated. Also, if there was a cookie with a session ID, if this ID is no longer valid (because it has expired, for example, or is a remnant from a very old interaction and doesn't exist in the current cache of session IDs) CherryPy also generates a new session ID.

At this point, no persistent information is stored if this is a new session, but if it's an existing session there might be persistent data available. If there is, CherryPy creates a `Session` object and initializes it with the available persistent data. If not, it creates an empty `Session` object. This object is available as a global variable `cherrypy.session`.

The next step for CherryPy is to pass control to the function that will handle the request. This handler has access to the `Session` object and may change it, for example, by storing additional information for later reuse. (Note that the `Session` object acts like a dictionary so you can simply associate values with keys with `cherrypy.session['key']=value`. The only restriction to the keys and values is that they must be serializable if the persistent storage is on disk).

Then before returning the results generated by the handler, CherryPy checks if the `Session` object has changed. If (and only if) it has, are the contents of the `Session` object saved to a more permanent storage.

Finally, the response is returned accompanied by a cookie with the session ID.

Time for action – creating a logon screen

Our first task is to create a small application that does little more than present the user with a logon screen. It will be the starting point of our tasklist application and many others as well.

The code for this example as well as most other examples in this book is available from the Packt website. If you have not downloaded it yet, this might be a good time to do so.

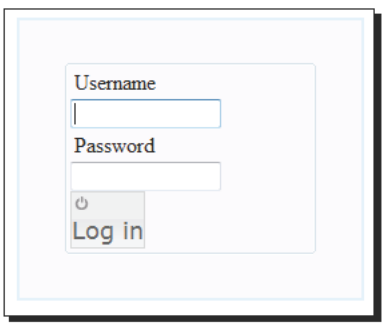
Enter the following pieces of code and save it in a file called `logonapp.py` in the same directory as the other files distributed with this chapter (*Chapter 3* in the sample code):

Chapter3/logonapp.py

```
import cherrypy
import logon
class Root(object):
    logon = logon.Logon(path="/logon",
                        authenticated="/",
                        not_authenticated="/goaway")
    @cherrypy.expose
    def index(self):
           username=logon.checkauth('/logon')
           return '''
         <html><body>
         <p>Hello user <b>%s</b></p>
         </body></html>'''%username
    @cherrypy.expose
    def goaway(self):
           return '''
       <html>
```

```
            <body><h1>Not authenticated, please go away.</h1>
            </body></html>'''

        @cherrypy.expose
        def somepage(self):
                username=logon.checkauth('/logon',returntopage=True)
                return '''<html>
                            <body><h1>This is some page.</h1>
                            </body>
                            </html>'''

    if __name__ == "__main__":
        import os.path
        current_dir = os.path.dirname(os.path.abspath(__file__))

        cherrypy.quickstart(Root(),config={
                '/': {'tools.sessions.on': True }
                }
                )
```

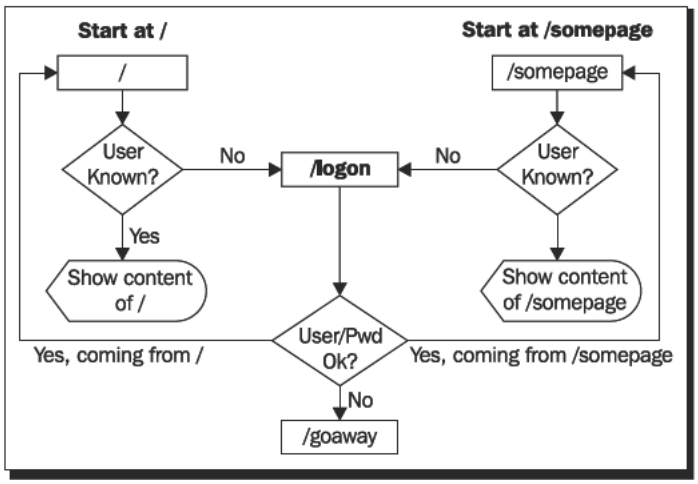
If you now run `logonapp.py`, a very simple application is available on port 8080. It presents the user with a logon screen when the top level page `http://localhost:8080/` is accessed. An example is shown in the following illustration:



If a correct username/password combination is entered, a welcome message is shown. If an unknown username or wrong password is entered, the user is redirected to `http://localhost:8080/goaway`.

The `somepage()` method (highlighted) returns a page with (presumably) some useful content. If the user is not yet authenticated, the logon screen is shown and upon entering the correct credentials, the user is directed back to `http://localhost:8080/somepage`.

The complete tree of web pages within the logon sample application and the possible paths the user may pick through is shown next:



Logon + session ID vs. HTTP basic authentication

You may wonder why we choose not to reuse CherryPy's bundled `auth_basic` tool that offers basic authentication (for more information on this tool, see `http://www.cherrypy.org/wiki/BuiltinTools#tools.auth_basic`). If all we wanted was to check whether a user is allowed access to a single page, this would be a good choice. The basic authentication is sufficient to authenticate a user, but has no concept of a session. This means we lack a way to store data that needs to be accessible when we process subsequent requests by the same user. The `sessions` tool we use here does provide this additional functionality.

What just happened?

Part of the magic of `logonapp.py` is achieved by enabling the 'sessions' tool in CherryPy. This is what is done by passing the `tools.sessions.on` key with `True` as a value to the configuration dictionary for the `quickstart()` function.

However, most of the hard work in `logonapp.py` is actually performed by the module `logon`:

Chapter3/logon.py

```
import cherrypy
import urllib.parse

def checkauth(logonurl="/", returntopage=False):
    returnpage=''
```

```
    if returntopage:
           returnpage='?returnpage='
                  + cherrypy.request.script_name
                  + cherrypy.request.path_info
    auth = cherrypy.session.get('authenticated',None)
    if auth == None :
        raise cherrypy.HTTPRedirect(logonurl+returnpage)
    return auth
class Logon:
    base_page = '''
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<script type="text/javascript" src="/jquery.js" ></script>
<script type="text/javascript" src="/jquery-ui.js" ></script>
<style type="text/css" title="currentStyle">
    @import "/jquerytheme.css";
    @import "/static/css/logon.css";
</style>
</head>
<body id="logonscreen">
<div id="content">
%s
</div>
<script type="text/javascript">$("button").button({icons: {primary:
'ui-icon-power'}})</script>
</body>
</html>
'''
    logon_screen = base_page % '''
<form class="login" action="%s/logon" method="GET">
<fieldset>
<label for="username">Username</label>
<input id="username" type="text" name="username" />
<script type="text/javascript">$("#username").focus()</script>
<label for="password">Password</label>
<input id="password" type="password" name="password" />
<input type="hidden" name="returnpage" value="%s" />
<button type="submit" class="login-button" value="Log in">
Log in
</button>
</fieldset>
```

```
    </form>
    '''
        not_authenticated =
            base_page % '''<h1>Login or password not correct</h1>'''
        def __init__(self, path="/logon",
                           authenticated="/", not_authenticated="/"):
            self.path=path
            self.authenticated=authenticated
            self.not_authenticated=not_authenticated
        @staticmethod
        def checkpass(username,password):
            if username=='user' and password=='secret': return True
            return False
        @cherrypy.expose
        def index(self,returnpage=''):
            return Logon.logon_screen % (
                           self.path,urllib.parse.quote(returnpage))
        @cherrypy.expose
        def logon(self,username,password,returnpage=''):
            returnpage = urllib.parse.unquote(returnpage)
            if Logon.checkpass(username,password):
                  cherrypy.session['authenticated']=username
                  if returnpage != '':
                        raise cherrypy.InternalRedirect(returnpage)
                  else:
                        raise cherrypy.InternalRedirect(
                                                self.authenticated)
            raise cherrypy.InternalRedirect(self.not_authenticated)
        @cherrypy.expose
        def logoff(self,logoff):
            cherrypy.lib.sessions.expire()
            cherrypy.session['authenticated']=None
            raise cherrypy.InternalRedirect(self.not_authenticated)
```

The logon module implements a utility function `checkauth()` (highlighted). This function is designed to be called from anywhere inside a CherryPy application. If the user is already authenticated, it will return the username; otherwise it will redirect the user to a URL that should present the user with a logon screen. If the `returnpage` parameter is true, this URL is augmented with an extra parameter `returnpage` containing the URL of the page that invoked `checkauth()`. The logon page (or rather the handler implementing it) should be designed to redirect the user to the URL in this parameter if the authentication is successful.

As we have seen, typical use for the `checkauth()` function would be to call it from every page handler that serves content that requires authentication.

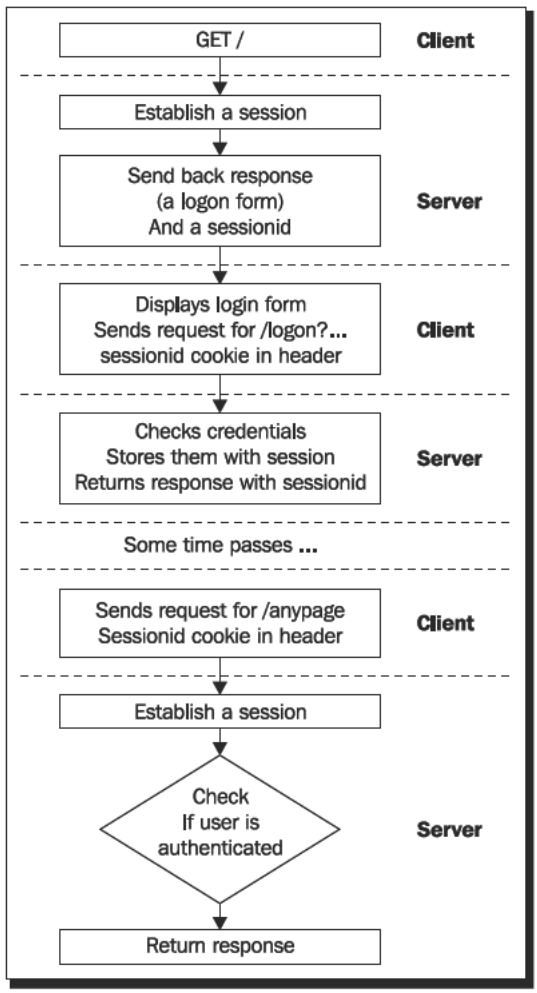
`checkauth()` itself does just two things: First it determines the page to return to (if necessary) by concatenating the `script_name` and `path_info` attributes from the `cherrypy.request` object that CherryPy makes available. The first one contains the path where a CherryPy tree is mounted, the last one contains the path within that tree. Together they form the complete path to the handler that invoked this `checkauth()` function.

The second thing that `checkauth()` does is it determines whether `cherrypy.session` (a dictionary like Session object) contains an `authenticated` key. If it does, it returns the associated value, if not, it redirects to the logon page.

The `cherrypy.session` variable is a `cherrypy.lib.sessions.Session` object available to each request. It acts like a dictionary and initially it is devoid of any keys. When a value is assigned to the first new key, a persistent object is created that is associated with the session ID and upon finishing a request, the `Session` object is stored and its session ID is passed as the value of a `session_id` cookie in the response headers. If a subsequent request contains a request header with a `session_id` cookie, a `Session` object with the corresponding session ID is retrieved from storage, making any saved key/value pairs available again.

The default storage scheme is to keep data in memory. This is fast and simple but has the disadvantage that restarting the CherryPy server will discard this data, effectively expiring all sessions. This might be ok for short-lived sessions, but if you need a more persistent solution, it is possible to store the session information as files (by setting the `tools.sessions.storage_type` configuration key to `"file"`) or even to a database backend. For more about sessions, see CherryPy's online documentation on the subject at `http://cherrypy.org/wiki/CherryPySessions`.

The various steps in the communication between the client and the server during a session are shown in the following illustration:



The bulk of the `logon` module is provided by the `Logon` class. It implements several methods (these methods are highlighted in the code listed on the previous pages as well):

- `__init__()` will initialize a `Logon` instance to hold the path to the point where this `Logon` instance is mounted on the tree of handlers, together with the default URLs to redirect to successful and unsuccessful authentication.
- `checkpass()` is a static function that takes a username and a password and returns `True` if these are a matching pair. It is designed to be overridden by a more suitable definition.

`Logon` also exposes three handler methods to the CherryPy engine:

- `index()` is a method that will serve the actual logon screen
- `logon()` is passed the username and password when the user clicks on the logon button
- `logoff()` will expire a session, causing subsequent calls to `checkauth()` to redirect the user to the logon screen

The `Logon` class also contains a number of class variables to hold the HTML presented by the `index()` method. Let's look at the methods in detail.

And what about security? The `Logon` class we design here has no facilities to prevent people from eavesdropping if they have access to the wire that transports the HTTP traffic. This is because we transmit the passwords unencrypted. We may implement some sort of encryption scheme ourselves, but if your design requires some form of protection, it is probably better and easier to communicate over a secure HTTPS channel. CherryPy may be configured to use HTTPS instead of HTTP. More on it can be found at: `http://cherrypy.org/wiki/ServerObject`.

Pop quiz – session IDs

1. If the client sends a new session ID again and again, wouldn't that fill up all storage on the server eventually?
2. If the client has cookies disabled, what happens to the generation of session IDs?
    a. The server will stop generating new session IDs, returning the same ID repeatedly
    b. The server will stop returning new session IDs
    c. The server will keep generating and returning new session IDs

Serving a logon screen

The `index()` method serves the HTML to present the user with a logon screen. At its core, this HTML is a `<form>` element with three `<input>` elements: a regular text input where the user may enter his/her username, a password input (that will hide the characters that are entered in this field), and an `<input>` element that has a `hidden` attribute. The `<form>` element has an `action` attribute that holds the URL of the script that will process the variables in the form when the user clicks the logon button. This URL is constructed to point to the `logon()` method of our `Logon` class by appending `/logon` to the path that the `Logon` instance was mounted on in the CherryPy tree.

The `<input>` element we marked as hidden is initialized to hold the URL that the user will be redirected to when `logon()` authenticates the user successfully.

The form that makes up the logon screen also contains a tiny piece of JavaScript:

```
$("#username").focus()
```

It uses jQuery to select the input element that will receive the username and gives it focus. By placing the cursor in this field, we save the user the effort of pointing and clicking on the username field first before the username can be entered. Now he can start typing right away. Note that this code snippet is not placed near the end of the document, but right after the `<input>` element to ensure execution as soon as the `<input>` element is defined. The logon page is so small that this might be irrelevant, but on slow loading pages, key presses might be misdirected if we waited to shift the focus until the whole page had loaded.

Be aware that the logon form we construct here has a `<form>` element with an `action="GET"` attribute. This works fine, but has a disadvantage: parameters passed with a `GET` method are appended to the URL and may end up in the log files of the server. This is convenient when debugging, but you might not want that for a production environment, as this might leave passwords exposed. The `action` attribute can be changed to `POST` though without any change to the Python code handling the request as CherryPy takes care of the details. Parameters passed to a `POST` method are not logged, so a `POST` method might be better suited to a password verification request.

Setting up a session

The `logon()` method is passed the contents of all the `<input>` elements in the form as parameters. The `username` and `password` parameters are passed to the `checkpass()` method and if the user's credentials are right, we establish a session by associating the username with the authenticated key in our session storage with `cherrypy.session['authenticated']=username`.

This will have the effect that every response sent to the browser will contain a cookie with a session ID and any subsequent request to CherryPy that contains this cookie again will cause the handler for that request to have access to this same session storage.

After successful authentication, `logon()` redirects the user to the return page if one was passed to it or to the default page passed to it upon initialization of the `Logon` instance. If authentication fails, the user is redirected to a non-authorized page.

Expiring a session

The `logoff()` method is provided to offer a possibility to actively expire a session. By default, a session expires after 60 minutes, but the user might want to sign off explicitly, either to make sure that no one sneaks behind his keyboard and continues in his name or to log on as a different persona. Therefore, you will find, in most applications, a discrete logoff button, often positioned in the upper-right corner. This button (or just a link) must point to the URL that is handled by the `logoff()` method and will cause the session to be invalidated immediately by removing all session data.

Note that we have to take special precautions to prevent the browser from caching the response from the `logoff()` method, otherwise it may simply redisplay the response from the last time the logoff button was pressed without actually causing `logoff()` to be called. Because `logoff()` always raises an `InternalRedirect` exception, the actual response comes from a different source. This source, for example, the `goaway()` method in the `Root` class must be configured to return the correct response headers in order to prevent the web browser from caching the result. This is accomplished by configuring the `goaway()` method in `logonapp.py` with CherryPy's expires tool like the following:

Chapter3/logonapp.py

```
@cherrypy.expose
    def goaway(self):
            return '''
<html><body>
<h1>Not authenticated, please go away.</h1>
</body></html>
'''
    goaway._cp_config = {

            'tools.expires.on':True,
            'tools.expires.secs':0,
            'tools.expires.force':True}
```

The highlighted line is where we configure the handler (the `goaway()` method) to set expiration headers in the response by assigning a configuration dictionary to the `_cp_config` variable.

Assigning to a variable that is part of a function might seem odd, but functions and methods in Python are just objects and any object may have variables. New variables might be assigned to an object even after its definition. Upon calling a handler, CherryPy checks if that handler has a `_cp_config` variable and acts accordingly. Note that the `@cherrypy.expose` decorator also merely sets the `expose` variable on the handler to `true`.

Have a go hero – adding a logon screen to the spreadsheet application

In the previous chapter, we had created an application that serves a spreadsheet. If you wanted to serve this spreadsheet only to authenticated users, what would we have to change to use the logon module presented in the previous section?

Hint: You need to do three things, one involves mounting an instance of the `Logon` class on the CherryPy tree, the other is changing the handler that serves the spreadsheet to check for authentication, and finally you need to enable sessions.

An example implementation is available as `spreadsheet3.py`.

Designing a task list

Now that we have looked at ways to authenticate the users, let's look at the implementation of the task list itself.

A task list would be unusable if its contents evaporated once the browser was closed. We therefore need some way to persistently store these task lists. We could use a database and many of the example applications in this book do use a database to store data. For this application, we will opt to use the filesystem as a storage medium, simply storing tasks as files containing information about a task, with separate directories for each user. If we dealt with huge amounts of users or very long task lists, the performance of such an implementation probably wouldn't suffice, but by using simple files for storage, we won't have to design a database schema which saves us quite some time.

By limiting ourselves to fairly short task lists, our user interface may be kept relatively simple as there will be no need for pagination or searching. This doesn't mean the user interface shouldn't be easy to use! We will incorporate jQuery UI's **datepicker** widget to assist the user with choosing dates and will add tooltips to user interface components to provide a shallow learning curve of our task list application.

The final requirements more or less define what we understand a task to be and what we are supposed to do with it: A task has a description and a due date and because it can be marked as done, it should be able to store that fact as well. Furthermore, we limit this application to adding and deleting tasks. We explicitly do not provide any way to alter a task, except for marking it as done.

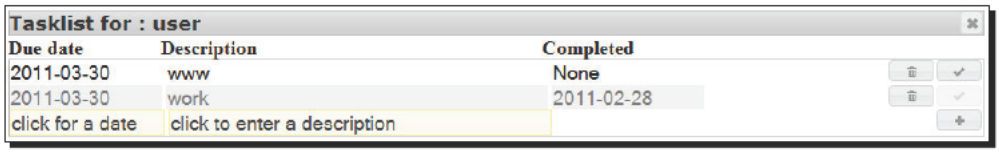
Time for action – running tasklist.py

Let's first have a look at what the application looks like:

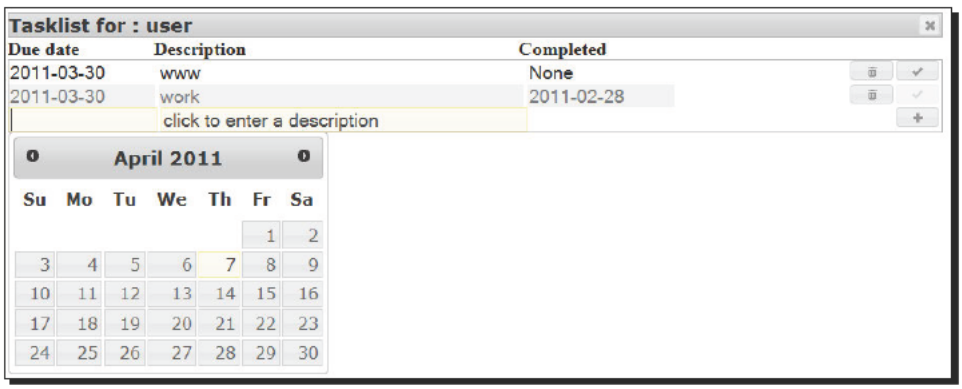
1. Start up `tasklist.py` from the code directory of this chapter.
2. Point your browser to `http://localhost:8080`.

3. In the logon screen, enter **user** as the username and **secret** as the password.

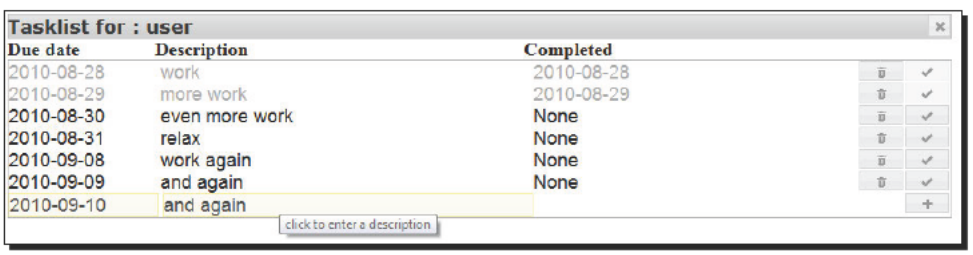
4. You are now presented with a rather stark looking and empty task list:



You should be able to add a new task by entering a date and a description in the input boxes and pressing the add button. Entering a date is facilitated by jQuery UI's datepicker widget that will pop up once you click the input field for the date, as shown in the following screenshot:



Once you have added one or more tasks, you can now either delete those tasks by clicking the button with the little trash can icon or mark it as done by clicking the button with the check icon. Tasks marked as done have a slightly different background color depending on the chosen theme. If you mark a task as done, its completion date will be today. You can select a different date by clicking on the completion date of a task (displayed as **None** for an unfinished task). It will present you with yet another datepicker, after which the selected date will be stored as the completion date once the done button is clicked. The following screenshot gives an impression of a task list with numerous items:



There is some hidden magic that might not be immediately obvious. First of all, all the tasks are sorted according to their **Due date**. This is done on the client-side with the help of some JavaScript and a jQuery plugin, as we will see in the section on JavaScript. Also accomplished with some JavaScript are the tooltips. Both hovering tooltips on every button and the inline help text inside the `<input>` elements are added with the same script. We will examine this in depth.

What just happened?

`tasklist.py` is rather straightforward as it delegates most work to two modules: the `logon` module that we encountered in the previous sections and a `task` module that deals with displaying and manipulating task lists.

The highlighted line in the following code shows the core of the application. It starts up CherryPy with a suitable configuration. Note that we enabled the sessions tool, so that we can actually use the `logon` module. Also, we construct the path to jQuery UI's theme stylesheet in such a way that it depends on the `theme` variable to make changing the application's theme simple (second highlight).

The instance of the `Root` class that we pass to `quickstart()` creates a simple tree:

```
/
/logon
/logon/logon
/logon/logoff
/task
/task/add
/task/mark
```

The top level URL `/` returns the same content as `/login` by calling the `index()` method of the `Logon` instance. We could have used an `InternalRedirect` exception, but this is just as simple. The paths starting with `/task` are all handled by an instance of the `Task` class:

Chapter3/tasklist.py

```
import cherrypy
import os.path
import logon
import task
current_dir = os.path.dirname(os.path.abspath(__file__))
theme = "smoothness"
class Root(object):
    task = task.Task(logoffpath="/logon/logoff")
    logon = logon.Logon(path="/logon",
```

```
                        authenticated="/task",
                        not_authenticated="/")
    @cherrypy.expose
    def index(self):
          return Root.logon.index()
if __name__ == "__main__":
    cherrypy.quickstart(Root(),config={
          '/':
          { 'log.access_file':os.path.join(current_dir,"access.log"),
          'log.screen': False,
          'tools.sessions.on': True
          },
          '/static':
          { 'tools.staticdir.on':True,
          'tools.staticdir.dir':os.path.join(current_dir,"static")
          },
          '/jquery.js':
          { 'tools.staticfile.on':True,
          'tools.staticfile.filename':os.path.join(current_dir,
          "static","jquery","jquery-1.4.2.js")
          },
          '/jquery-ui.js':
          { 'tools.staticfile.on':True,
          'tools.staticfile.filename':os.path.join(current_dir,
          "static","jquery","jquery-ui-1.8.1.custom.min.js")
          },
          '/jquerytheme.css':
          { 'tools.staticfile.on':True,
          'tools.staticfile.filename':os.path.join(current_dir,
          "static","jquery","css",theme,"jquery-ui-1.8.4.custom.css")
          },
          '/images':
          { 'tools.staticdir.on':True,
          'tools.staticdir.dir':os.path.join(current_dir,
          "static","jquery","css",theme,"images")
          }
    })
```

Python: the task module

The `task` module is implemented in the file `task.py`. Let's look at the parts that make up this file.

Time for action – implementing the task module

Have a look at the Python code in `task.py`:

Chapter3/task.py

```
import cherrypy
import json

import os
import os.path
import glob
from configparser import RawConfigParser as configparser
from uuid import uuid4 as uuid
from datetime import date

import logon
```

This first part illustrates Python's "batteries included" philosophy nicely: besides the `cherrypy` module and our own `logon` module, we need quite a bit of specific functionality. For example, to generate unique identifiers, we use the `uuid` module and to manipulate dates, we use the `datetime` module. All of this functionality is already bundled with Python, saving us an enormous amount of development time. The next part is the definition of the basic HTML structure that will hold our task list:

Chapter3/task.py

```
base_page = '''
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
<script type="text/javascript" src="/jquery.js" ></script>
<script type="text/javascript" src="/jquery-ui.js" ></script>
<style type="text/css" title="currentStyle">
   @import "/static/css/tasklist.css";
   @import "/jquerytheme.css";
</style>
<script type="text/javascript" src="/static/js/sort.js" ></script>
<script type="text/javascript" src="/static/js/tooltip.js" ></script>
<script type="text/javascript" src="/static/js/tasklist.js" ></script>
</head>
<body id="%s">
<div id="content">
%s
</div>
</body>
```

```
</html>
'''
```

Again the structure is simple, but besides the themed stylesheet needed by jQuery UI (and reused by the elements we add to the page), we need an additional stylesheet specific to our task list application. It defines specific layout properties for the elements that make up our task list (first highlight). The highlighted `<script>` elements show that besides the jQuery and jQuery UI libraries, we need some additional libraries. Each of them deserves some explanation.

What just happened?

The first JavaScript library is `sort.js`, a code snippet from James Padolsey (`http://james.padolsey.com/tag/plugins/`) that provides us with a plugin that allows us to sort HTML elements. We need this to present the list of tasks sorted by their due date.

The second is `tooltip.js` that combines a number of techniques from various sources to implement tooltips for our buttons and inline labels for our `<input>` elements. There are a number of tooltip plugins available for jQuery, but writing our own provides us with some valuable insights so we will examine this file in depth in a later section.

The last one is `tasklist.js`. It employs all the JavaScript libraries and plugins to actually style and sort the elements in the task list.

The next part of `task.py` determines the directory we're running the application from. We will need this bit of information because we store individual tasks as files located relative to this directory. The `gettaskdir()` function takes care of determining the exact path for a given username (highlighted). It also creates the `taskdir` directory and a sub directory with a name equal to username, if these do not yet exist with the `os.makedirs()` function (notice the final 's' in the function name: this one will create all intermediate directories as well if they do not yet exist):

Chapter3/task.py

```
current_dir = os.path.dirname(os.path.abspath(__file__))

def gettaskdir(username):
    taskdir = os.path.join(current_dir,'taskdir',username)
    # fails if name exists but is a file instead of a directory
    if not os.path.exists(taskdir):
           os.makedirs(taskdir)
    return taskdir
```

The `Task` class is where the handlers are defined that CherryPy may use to show and manipulate the task list. The `__init__()` method stores a path to a location that provides the user with a possibility to end a session. This path is used by other methods to create a suitable logoff button.

The `index()` method will present the user with an overview of all his/her tasks plus an extra line where a new task can be defined. As we have seen, each task is adorned with buttons to delete a task or mark it as done. The first thing we do is check whether the user is authenticated by calling the `checkauth()` function from our `logon` module (highlighted). If this call returns, we have a valid username, and with that username, we figure out where to store the tasks for this user.

Once we know this directory, we use the `glob()` function from the Python `glob` module to retrieve a list of files with a `.task` extension. We store that list in the `tasklist` variable:

Chapter3/task.py

```
class Task(object):
    def __init__(self,logoffpath="/logoff"):
            self.logoffpath=logoffpath
    @cherrypy.expose
    def index(self):
            username = logon.checkauth()
            taskdir = gettaskdir(username)
            tasklist = glob.glob(os.path.join(taskdir,'*.task'))
```

Next, we create a `tasks` variable that will hold a list of strings that we will construct when we iterate over the list of tasks. It is initialized with some elements that together form the header of our task list. It contains, for example, a small form with a logoff button and the headers for the columns above the list of tasks. The next step is to iterate over all files that represent a task (highlighted) and create a form with suitable content together with delete and done buttons.

Each `.task` file is structured in a way that is consistent with Microsoft Windows `.ini` files. Such files can be manipulated with Python's `configparser` module. The `.task` file is structured as a single `[task]` section with three possible keys. This is an example of the format:

```
[task]
description = something
duedate = 2010-08-26
completed = 2010-08-25
```

When we initialize a `configparser` object, we pass it a dictionary with default values in case any of these keys is missing. The `configparser` will read a file when we pass an open file descriptor to its `readfp()` method. The value associated with any key in a given section may then be retrieved with the `get()` method that will take a section and a key as parameters. If the key is missing, it supplies the default if that was provided upon initialization. The second highlighted line shows how this is used to retrieve the values for the `description` key.

Next, we construct a form for each `.task` file. It contains read-only `<input>` elements to display the **Due date**, **Description**, and the completion date plus buttons to delete the task or mark it as done. When these buttons are clicked the contents of the form are passed to the `/task/mark` URL (handled by the `mark()` method). The method needs to know which file to update. Therefore, it is passed a hidden value: the basename of the file. That is, the filename without any leading directories and stripped of its `.task` extension:

Chapter3/task.py

```
            tasks = [
'''
<div class="header">
Tasklist for user <span class="highlight">%s</span>
  <form class="logoff" action="%s" method="GET">
    <button type="submit" name="logoffurl"
            class="logoff-button" value="/">Log off
    </button>
  </form>
</div>
'''%(username,self.logoffpath),
'''
<div class="taskheader">
  <div class="left">Due date</div>
  <div class="middle">Description</div>
  <div class="right">Completed</div>
</div>
''','<div id="items" class="ui-widget-content">']
            for filename in tasklist:
                d = configparser(
                defaults={'description':'',
                          'duedate':'',
                          'completed':None})
                id = os.path.splitext(os.path.basename(filename))[0]
                d.readfp(open(filename))
                description = d.get('task','description')
                duedate = d.get('task','duedate')
                completed = d.get('task','completed')
                tasks.append(
'''
<form class="%s" action="mark" method="GET">
  <input type="text" class="duedate left"
         name="duedate" value="%s" readonly="readonly" />
  <input type="text" class="description middle"
         name="description" value="%s" readonly="readonly" />
```

```
    <input type="text" class="completed right editable-date tooltip"
           title="click to select a date, then click done"
           name="completed" value="%s" />
    <input type="hidden" name="id" value="%s" />
    <button type="submit" class="done-button"
            name="done" value="Done" >Done
    </button>
    <button type="submit" class="del-button"
            name="delete" value="Del" >Del
    </button>
  </form>
  '''%('notdone' if completed==None else 'done',
       duedate,description,completed,id))
            tasks.append(
  '''
  <form class="add" action="add" method="GET">
    <input type="text" class="duedate left editable-date tooltip"
           name="duedate" title="click to select a date" />
    <input type="text" class="description middle tooltip"
           title="click to enter a description" name="description"/>
    <button type="submit" class="add-button"
            name="add" value="Add" >Add
    </button>
  </form>
  </div>
  ''')
            return base_page%('itemlist',"".join(tasks))
```

Finally, we append one extra form with the same type of input fields for **Due date** and **Description** but this time, not marked as read-only. This form has a single button that will submit the contents to the `/task/add` URL. These will be handled by the `add()` method. The actual content returned by the `index()` method consists of all these generated lines joined together and embedded in the HTML of the `base_page` variable.

Adding new tasks

New tasks are created by the `add()` method. Besides the value of the add button (which is not relevant), it will take a `description` and a `duedate` as parameters. To prevent accidents, it first checks if the user is authenticated, and if so, it determines what the `taskdir` for this user is.

We are adding a new task so we want to create a new file in this directory. To guarantee that it has a unique name, we construct this filename from the path to this directory and a globally unique ID object provided by Python's `uuid()` function from the `uuid` module. The `.hex()` method of a `uuid` object returns the ID as a long string of hexadecimal numbers that we may use as a valid filename. To make the file recognizable to us as a task file, we append the `.task` extension (highlighted).

Because we want our file to be readable by a `configparser` object, we will create it with a `configparser` object to which we add a `task` section with the `add_section()` method and `description` and `duedate` keys with the `set()` method. Then we open a file for writing and use the open file handle to this file within a context manager (the `with` clause), thereby ensuring that if anything goes wrong when accessing this file, it will be closed and we will proceed to redirect the user to that list of tasks again. Note that we use a relative URL consisting of a single dot to get us the index page. Because the `add()` method handles a URL like `/task/add` redirecting to '.' (the single dot), will mean the user is redirected to `/task/`, which is handled by the `index()` method:

Chapter3/task.py

```
@cherrypy.expose
def add(self,add,description,duedate):
       username = logon.checkauth()
       taskdir = gettaskdir(username)
       filename = os.path.join(taskdir,uuid().hex+'.task')
       d=configparser()
       d.add_section('task')
       d.set('task','description',description)
       d.set('task','duedate',duedate)
       with open(filename,"w") as file:
              d.write(file)
       raise cherrypy.InternalRedirect(".")
```

Deleting a task

Deleting or marking a task as done are both handled by the `mark()` method. Besides an ID (the basename of an existing `.task` file), it takes `duedate`, `description`, and `completed` parameters. It also takes optional `done` and `delete` parameters, which are set depending on whether the done or delete buttons are clicked respectively.

Again, the first actions are to establish whether the user is authenticated and what the corresponding task directory is. With this information, we can construct the filename we will act on. We take care to check the validity of the `id` argument. We expect it to be a string of hexadecimal characters only and one way to verify this is to convert it using the `int()` function with 16 as the base argument. This way, we prevent malicious users from passing a file path to another user's directory. Even though it is unlikely that a 32 character random string can be guessed, it never hurts to be careful.

The next step is to see if we are acting on a click on the done button (highlighted in the following code). If we are, we read the file with a `configparser` object and update its `completed` key.

The `completed` key is either the date that we were passed as the `completed` parameter or the current date if that parameter was either empty or `None`. Once we have updated the `configparser` object, we write it back again to the file with the `write()` method.

Another possibility is that we are acting on a click on the delete button; in that case, the `delete` parameter is set. If so, we simply delete the file with the `unlink()` function from Python's `os` module:

Chapter3/task.py

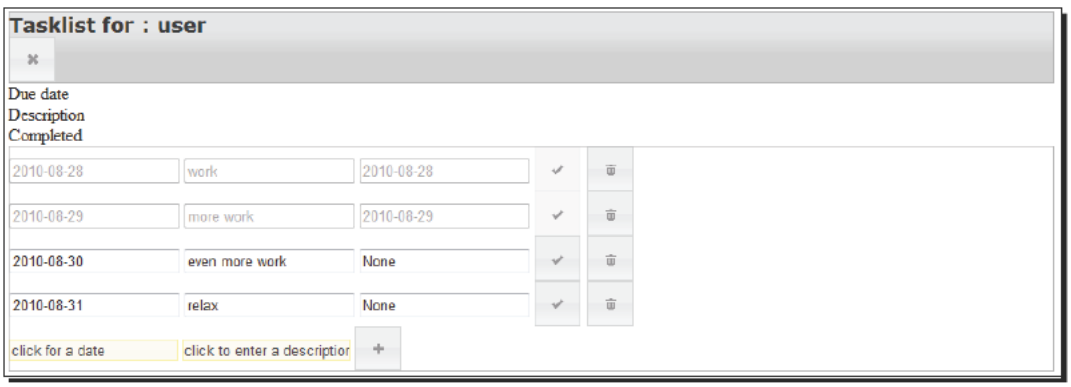
```
@cherrypy.expose
def mark(self,id,duedate,description,
             completed,done=None,delete=None):
       username = logon.checkauth()
       taskdir = gettaskdir(username)
       try:
              int(id,16)
       except ValueError:
              raise cherrypy.InternalRedirect(self.logoffpath)
       filename = os.path.join(taskdir,id+'.task')
       if done=="Done":
              d=configparser()
              with open(filename,"r") as file:
                     d.readfp(file)
              if completed == ""  or completed == "None":
                     completed = date.today().isoformat()
              d.set('task','completed',completed)
              with open(filename,"w") as file:
                     d.write(file)
       elif delete=="Del":
              os.unlink(filename)
       raise cherrypy.InternalRedirect(".")
```

JavaScript: tasklist.js

The buttons we present the end user need to be configured to respond to clicks in an appropriate manner and it would be nice if these buttons showed some intuitive icons as well. This is what we will take care of in `tasklist.js`.

Time for action – styling the buttons

The work done by `tasklist.js` is mainly concerned with styling the `<button>` elements and adding tooltips and inline labels to `<input>` elements. The results so far are shown in the following screenshot:



What just happened?

As can be seen in the first line of `tasklist.js` (code starts on the next page), the work to be done is scheduled after loading the complete document by passing it to jQuery's `$(document).ready()` function.

The first step is to add to any element with a `header` class the `ui-widget` and `ui-widget-header` classes as well. This will cause these elements to be styled in a way that is consistent with the chosen theme.

Then we configure the add button (or rather any element with the `add-button` class) as a jQuery UI button widget. The option object passed to it will configure it to show no text, but just a single icon depicting a thick plus sign. We also add an extra function to the click handler of the button that checks any element marked with the `inline-label` class to see if its contents are identical to the contents of its title attribute. If that is the case, we set the contents to the empty string, as this indicates that the user hasn't filled in anything in this element and we do not want to store the text of the inline label as the content of our new task (more about this in the section on tooltips). Note that we do nothing to prevent propagation of the click event, so if this button is of the `submit` type (and our add button is) the `submit` action will still be performed.

All elements with the `del-button` class (highlighted) are then styled with an icon of a trash can. The buttons also receive an extra click handler that will remove the disabled attribute from their siblings (the input fields in the same form) to make sure the submit action will receive the contents even from fields that are marked as disabled.

Next, the other `<button>` elements are adorned with an appropriate icon and to any text or password `<input>` element we add a `textinput` class to mark it for the tooltip library.

In the second highlighted line, we encounter jQuery UI's datepicker widget. The datepicker widget greatly simplifies entering dates for the user and is now more or less a staple item in any web application or website that asks the user to enter a date. jQuery UI's datepicker is very straightforward to use, yet comes with a host of configuration options (all of them documented at `http://jqueryui.com/demos/datepicker/`).

We use the `dateFormat` option to configure the datepicker to store dates as YYYY-MM-DD. Datepicker has a number of predefined formats and this one happens to be an international standard as well as a suitable format to sort dates in a simple way. We also configure the datepicker to call a function when the user closes the datepicker. This function removes any `inline-label` class, preventing the newly entered date to appear in the colors associated with any inline label (as we see later, when we look at `tasklist.css`, we style the colors of any element with an `inline-label` class in a distinct way).

Earlier, we indicated that we wanted to present the list of tasks ordered by their due date. We therefore apply the `sort()` plugin from `sort.js` to all `<input>` elements with a `duedate` class. `sort()` takes two arguments. The first one is a comparison function that is passed two elements to compare. In our case, that will be `<input>` elements that contain a date in the YYYY-MM-DD format, so we can simply compare the values of these elements as text and return plus or minus one. The second argument is a function that takes no arguments and should return the element to be sorted. The input element with the due date is available as the `this` variable within this function and we use it to return the parent of the input element. This parent will be the `<form>` element that encloses it and because we represent each task as a form, we want those forms to be sorted, not just the `<input>` elements inside these forms.

The last set of actions in `tasklist.js` adds a `disabled` attribute to any `<input>` element within an element that has a `done` class and disables any done button. This will ensure that tasks marked as done cannot be altered:

Chapter3/tasklist.js

```
$(document).ready(function(){
    $(".header").addClass("ui-widget ui-widget-header");

    $(".add-button").button(
       {icons:{primary: 'ui-icon-plusthick' },
```

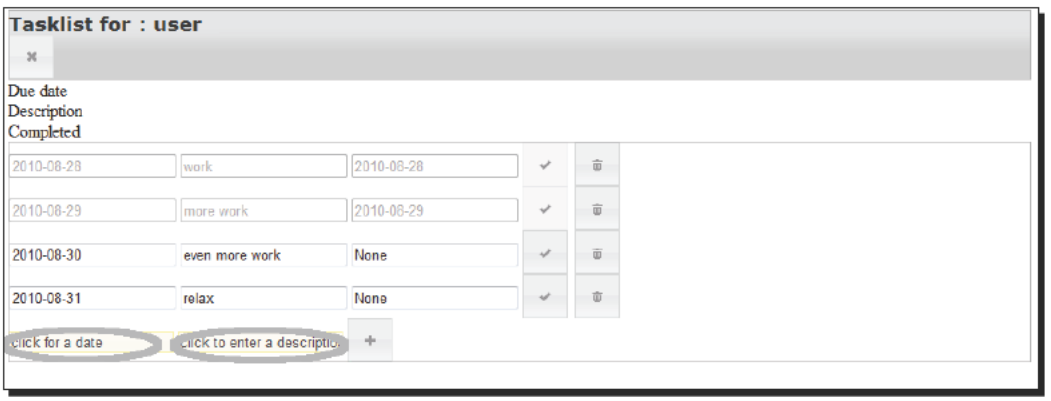
```
        text:false}).click(function(){
          $(".inline-label").each(function() {
                   if($(this).val() === $(this).attr('title')) {
                          $(this).val('');
                   };
          })
       });
      $(".del-button").button(
         {icons:{primary: 'ui-icon-trash' },
          text:false}).click(function(){
            $(this).siblings("input").removeAttr("disabled");
       });
      $(".done-button").button(  {icons: {primary:'ui-icon-check'},
                                  text:false});
      $(".logoff-button").button({icons: {primary:'ui-icon-closethick'},
                                  text:false});
      $(".login-button").button( {icons: {primary:'ui-icon-play'},
                                  text:false});
      $(":text").addClass("textinput");
      $(":password").addClass("textinput");
      $( ".editable-date" ).datepicker({
             dateFormat: $.datepicker.ISO_8601,
             onClose: function(dateText,datePicker){
             if(dateText != ''){$(this).removeClass("inline-label");}}
         });
      $("#items form input.duedate").sort(
             function(a,b){return $(a).val() > $(b).val() ? 1 : -1;},
             function(){ return this.parentNode; }).addClass(
                                                    "just-sorted");
      $(".done .done-button").button( "option", "disabled", true );
      $(".done input").attr("disabled","disabled");
});
```

JavaScript: tooltip.js

`tooltip.js` is a bit of a misnomer as its most interesting part is not about tooltips but inline labels. Inline labels are a way to convey helpful information not by means of a hovering tooltip, but by putting text inside text input elements. This text then disappears when the user clicks the input field and starts typing. There are many implementations to be found on the web, but the most clear and concise one I found is from `http://trevordavis.net/blog/tutorial/jquery-inline-form-labels/`.

Time for action – implementing inline labels

Take a look again at the screenshot of the list of tasks:



The highlighted parts show what we mean by inline labels. The input fields display some helpful text to indicate their use and when we click such a field, this text will disappear and we can enter our own text. If we abort the input by clicking outside the input field when we have not yet entered any text, the inline label is shown again.

What just happened?

`tooltip.js` shows a number of important concepts: First how to apply a function to each member of a selection (highlighted). In this case, we apply the function to all `<input>` elements that have a `title` attribute. Within the function passed to the `each()` method, the selected `<input>` element is available in the `this` variable. If the content of an `<input>` element is completely empty, we change its content to that of the `title` attribute and add the class `inline-label` to the `<input>` element. That way, we can style the text of an inline label differently than the regular input text if we like, for example, a bit lighter to make it stand out less.

The second concept shown is binding to the **focus** and **blur** events. When the user clicks an `<input>` element or uses the *Tab* key to navigate to it, it gains focus. We can act upon this event by passing a function to the `focus()` method. In this function, the `<input>` element that gains focus is again available in the `this` variable and we check if the content of this `<input>` element is equal to the content of its `title` attribute. If this is true, the user hasn't yet changed the content, so we empty this element by assigning an empty string to it (highlighted).

The same line shows another important concept in jQuery, that of **chaining**. Most jQuery methods (like `val()` in this example) return the selection they act upon, allowing additional methods to be applied to the same selection. Here we apply `removeClass()` to remove the `inline-label` class to show the text the user is typing in the regular font and color for this `<input>` element.

We also act on losing focus (commonly referred to as *blurring*), for example, when the user clicks outside the `<input>` element or uses the *Tab* key to navigate to another element. We therefore pass a function to the `blur()` method. This function checks whether the content of the `<input>` element is empty. If so, then the user hasn't entered anything and we insert the content of the `title` attribute again and mark the element with an `inline-label` class.

Chapter3/tooltip.js

```
$(document).ready(function() {
    $('input[title]').each(function() {
            if($(this).val() === '') {
                    $(this).val($(this).attr('title'));
                    $(this).addClass('inline-label');
            }
            $(this).focus(function() {
                    if($(this).val() === $(this).attr('title')) {
                            $(this).val('').removeClass('inline-label');
                    }
            });
            $(this).blur(function() {
                    if($(this).val() === '') {
                            $(this).val($(this).attr('title'));
                            $(this).addClass('inline-label');
                    }
            });
    });
});
```

CSS: tasklist.css

Without some additional styling to tweak the layout, our tasklist application would look a bit disheveled, as seen before.

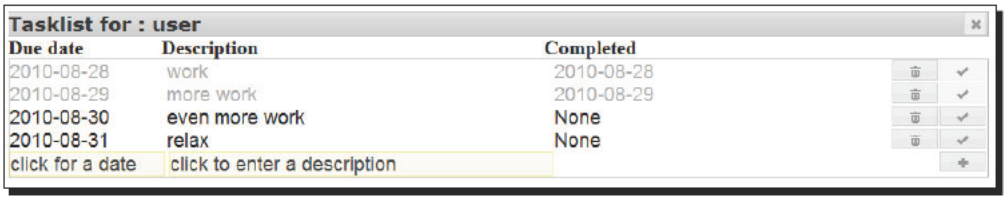
Our main challenges are aligning all columns and moving all buttons consistently to the right. All elements in our HTML markup that make up the columns are marked with a class to indicate that they belong in the left, middle, or right column. All we have to do to align these columns is to set their width based on their class (highlighted).

The largest part of the rest of `tasklist.css` is concerned with either floating elements to the right (like buttons) or to the left (containers, like the `<div>` element with the `id` attribute `content`). Most containers are not only floated to the left, but also explicitly set to a width of 100 percent to make sure they fill the element they are contained in themselves. This is not always necessary to position them correctly, but if we do not take care, the background color of the enclosing element might show if an element doesn't fill its enclosing element:

Chapter3/tasklist.css

```
input[type="text"] {
   font-size:1.1em;
   margin:0;
   border:0;
   padding:0;}
.left, .right { width: 8em; }
.middle { width: 20em;}
form {
   float:left;
   border:0;
margin:0;
padding:0;
   clear:both;
   width:100%; }
form.logoff{
float:right;
   border:0;
margin:0;
padding:0;
   clear:both;
width:auto;
   font-size:0.5em;}
#items { float:left; clear:both; width:100%; }
.header { width:100%; }
.taskheader, .header, #content{ float:left; clear:both;}
.taskheader div { float:left; font-size:1.1em; font-weight:bold;}
.logoff-button, .done-button, .del-button, .add-button { float:right;}
.done-button, .add-button, .del-button { width: 6em; height: 1.1em; }
#content { min-width:900px;}
```

Note that our stylesheet only deals with measurements and font sizes. Any coloring is applied by the chosen jQuery UI theme. With the styles applied, the application looks a fair bit tidier:



Pop quiz – styling screen elements

1. In `tasklist.js`, we explicitly configured all buttons to show just an icon without any text. But what if we wanted to show both an icon and some text, what would we do?
2. If we didn't set the width of the form that makes up a task explicitly to 100 percent, what would the biggest disadvantage be?

Have a go hero – changing the date format of a datepicker

To display the date as ISO 8701 (or YYYY-MM-DD) isn't everybody's idea of a readable date format. For many people, the default mm/dd/yy is far more readable. How would you change `tasklist.js` to display the tasks with this default date format? Hint: it isn't enough to leave out the `dateFormat` option when calling the `datepicker()` plugin, you also need to change the comparator function to sort the tasks in a suitable manner.

For the impatient or curious readers: a sample implementation is available as `tasklist2.js` (start up `tasklist2.py` to see the effect).

Have a go hero – serving a task list from a different URL

One way to measure how reusable a piece of code is, is by using it in a situation that you did not yet have in mind when you designed it. Of course, that doesn't mean our task module should be able to function as a control application for an automobile construction plant, but what if we would like it to be part of a larger suite of applications served from the same root? Would we have to change anything?

Say we want to serve the tasklist application from the URL `/apps/task` instead of `/task`, what would we have to change?

Hint: In CherryPy, you can create a tree of URLs by assigning object instances to class variables of the object instance that is passed to the `quickstart()` method.

A possible implementation can be found in `tasklistapp.py`.

Summary

We have learned a lot in this chapter about session management and storing persistent information on the server. Specifically, we saw how to design a tasklist application and implement a logon screen. What a session is and how this allows us to work with different users at the same time and how to interact with the server, and add or delete tasks. We also learned how to make entering dates attractive and simple with jQuery UI's datepicker widget and how to style button elements and provide tooltips and inline labels to input elements.

Now that you know a little bit more about storing data on the server, you might wonder if storing information in plain files on the server filesystem is the most convenient solution. In many cases, it isn't and a database might be more suitable—which is the topic of the next chapter.

4
Tasklist II: Databases and AJAX

In this chapter, we will refactor our tasklist application. It will use a database engine on the server to store items and will use jQuery's AJAX functionality to dynamically update the contents of the web application. On the server side, we will learn how to use Python's bundled SQLite database engine. On the presentation side, we will encounter jQuery UI's event system and will learn how to react to mouse clicks.

In this chapter, we shall:

- Learn some benefits of using a database engine
- Get familiar with SQLite, a database engine distributed with Python
- Implement a password database with SQLite
- Learn how to design and develop a database-driven tasklist application
- Implement a test framework
- Learn how to make a web application more responsive using AJAX calls
- See how to implement interactive applications without `<form>` elements

So let's get on with it...

The advantages of a database compared to a filesystem

Storing records on a filesystem as separate files might be simple but does have several drawbacks:

- You have to define your own interface for accessing these files and parsing their contents. This is much more serious than it may sound because it compels you to develop and test a lot of specific functionality that you would otherwise get more or less for free from an existing library
- Accessing single files is much slower than selecting records from a table in a database. That might be workable as long as you know which record you want (as is the case in our tasklist application) but it certainly isn't workable when you want to select records based on the value of some attribute. This would necessitate opening each and every file and checking whether some attribute matches your criteria. On a data collection of hundreds of items or more, this would be prohibitively slow
- Also, it is difficult to implement transactions. If we want to guarantee that a set of actions will either be successful as a whole or will be rolled back if some part of it doesn't succeed, we will have to implement very sophisticated code ourselves if we want to use files on a filesystem
- When using files on a filesystem, it is a nuisance to define and maintain relations between records, and although our tasklist application is about as simple as it gets, almost any other application has more than one logical object and relations between them, so this is a serious issue.

Choosing a database engine

There are many database engines available that can be accessed from Python, both commercial and open source (`http://wiki.python.org/moin/DatabaseInterfaces`). Choosing the right database is not a trivial task as it might not only depend on functional requirements, but also on performance, the available budget, and hard to define requirements like easy maintenance.

In the applications we develop in this book, we have chosen to use the SQLite database engine (`http://www.sqlite.org`) for a number of reasons. First, it is free and included in Python's standard distribution. This is important for people writing books because it means that everyone who is able to run Python has access to the SQLite database engine as well. However, this is not a toy database: as a matter of fact, SQLite is a database that is used in many smartphones and high-profile applications like Firefox to store things like configurations and bookmarks. Furthermore, it is reliable and robust and, on top of that, quite fast.

It does have some drawbacks as well: first of all, it uses its own dialect of SQL (the language used to interact with the database) but to be fair, most database engines use their own dialect.

More seriously, the focus of SQLite is on embedded systems, the most visible consequence of that is that it doesn't have facilities to limit user access to a subset of tables and columns. There is just a single file on the filesystem that holds the contents of the database and the access rights to the file are determined by the filesystem on which it resides.

The final issue is not so much a drawback as a point of serious attention: SQLite does not enforce types. In many databases, the type defined for column determines rigidly what you can store in that column. When a column is defined as an INTEGER, the database engine, in general, won't allow you to store a string or a boolean value, whereas, SQLite does. This isn't as strange as it sounds once you compare it with the way Python manages variables. In Python, it is perfectly valid to define a variable and assign an integer to it, and later assign a string to the same variable. A variable in Python is just like a column in SQLite; it is just a pointer to a value and that value is not simply the value itself but also has an explicitly associated type.

The combination of availability, reliability, and a type system closely resembling Python's native way of dealing with values makes SQLite a very suitable database engine in many applications, although specific applications may have requirements that may be better served by other database engines, like PostgreSQL or MySQL. The latter might be an attractive alternative if your application will run on a web server that already provides MySQL.

Database-driven authentication

Before we start designing a database-driven tasklist application, let's first familiarize ourselves with SQLite in the context of a seemingly much simpler set of requirements: storing username/password combinations in a database and refactoring the Logon class to interact with this database.

The functional requirements are deceptively simple: to verify whether a username/password combination is valid, all we have to do is verify that the username/password combination given is present in the table of usernames and passwords. Such a table consists of two columns, one named `username` and the other named `password`. As it is never a good idea to store a collection of passwords in plaintext, we encrypt the passwords with a hash function so even if the password database is compromised, the bad guys will have a difficult time retrieving the passwords. This means, of course, that we have to hash a given password with the same hash function before comparing it to the stored password for the username but that doesn't add much complexity.

What does add complexity is the fact that CherryPy is multi-threaded, meaning that CherryPy consists of multiple lightweight processes accessing the same data. And although the developers of SQLite maintain the opinion that threads are evil (`http://www.sqlite.org/faq.html#q6`), threads make perfect sense in situations where a lot of time in the application is spent on waiting. This certainly is the case in web applications that spend a lot of time waiting for network traffic to complete, even in this time of broadband connections. The most effective way of using this waiting time is to enable a different thread to serve another connection so more users might enjoy a better interactive experience.

Hash functions (or Cryptographic hash functions to be more specific) convert any input string to an output string of limited length in such a way that it is very unlikely that two input strings that are different produce the same output. Also, conversion from input to output is a one way operation or at least it will cost a large amount of computing power to construct the input from the output. There are many useful hash functions known, the most popular ones are available in Python's `hashlib` module. The specific hash function we use here is called `SHA1`.

More about hashing can be found in the Python documentation at `http://docs.python.org/py3k/library/hashlib.html`, or on Wikipedia at `http://en.wikipedia.org/wiki/Cryptographic_hash_function`.

However, in SQLite, the connection object cannot be shared among threads. This doesn't mean that that we cannot use SQLite in a multi-threaded environment (despite the evilness of threads), but it does mean we have to make sure that if we want to access the same SQLite database from different threads, each thread must use a connection object that is exclusively created for that thread.

Fortunately, it is quite easy to instruct CherryPy to call a function the moment it starts a new thread and let that function create a new connection to our database, as we will see in the next section. If we would employ many different threads, this might be wasteful because the connection objects use some memory, but with a few tens of threads this doesn't pose much of a problem (The default number of threads in CherryPy is 10 and can be configured with the `server.thread_pool` configuration option). If the memory consumption is a problem, there are alternative solutions available, for example, in the form of a separate worker thread that handles all database interaction or a small pool of such threads. A starting point for this might be `http://tools.cherrypy.org/wiki/Databases`.

Time for action – authentication using a database

To illustrate how to use database-driven user authentication, run `logondbapp.py`. It will present you with a logon screen very similar to the one shown in the previous chapter. You may enter the built-in username/password combination of admin/admin, after which you will be presented with a welcoming page.

In order to make this mini application work with the database-driven version of user authentication, all we have to do is replace the reference to an instance of the Logon class to one of the `LogonDB` class, as highlighted in the following code (the full code is available as `logondbapp.py`):

Chapter4/logondbdb.py

```
import cherrypy
import logondb
class Root(object):
  logon = logondb.LogonDB(path="/logon", authenticated="/", not_
authenticated="/goaway", db="/tmp/pwd.db")
  @cherrypy.expose
  def index(self):
    username=Root.logon.checkauth('/logon')
    return '<html><body><p>Hello user <b>%s</b></p></body></
html>'%username
  @cherrypy.expose
  def goaway(self):
    return '<html><body><h1>Not authenticated, please go away.</h1></
body></html>'
  goaway._cp_config = {'tools.expires.on':True,'tools.expires.
secs':0,'tools.expires.force':True}
  @cherrypy.expose
  def somepage(self):
    username=Root.logon.checkauth('/logon',returntopage=True)
    return '<html><body><h1>This is some page.</h1></body></html>'
if __name__ == "__main__":
  import os.path
  current_dir = os.path.dirname(os.path.abspath(__file__))
  root = Root()
  def connect(thread_index):
    root.logon.connect()
  cherrypy.engine.subscribe('start_thread', connect)
  cherrypy.quickstart(root,config={ ... } )
```

Another important difference with the previous implementation is the highlighted definition of a `connect()` function that should be called for each new thread that is started by CherryPy. It calls the `connect()` method of the `LogonDB` instance to create a database connection object unique for a given thread. We register this function with the `cherrypy.engine.subscribe()` function and make it call our `connect()` function at the start of each new thread CherryPy starts.

What just happened?

The database-centered version of our `Logon` class, `LogonDB` inherits a lot from `Logon`. Specifically, all HTML-related logic is reused. `LogonDB` does override the `__init__()` method to store a path to a database file and makes sure the database is initialized using the `initdb()` method, if it does not yet exist (highlighted). It also overrides the `checkpass()` method because this method must now verify the existence of a valid username/password pair against a database table.

Chapter4/logondb.py

```
import logon
import sqlite3
from hashlib import sha1 as hash
import threading
import cherrypy

class LogonDB(logon.Logon):
  def __init__( self,path="/logon", authenticated="/", not_
authenticated="/", db="/tmp/pwd.db"):
    super().__init__(path,authenticated,not_authenticated)
    self.db=db
    self.initdb()

  @staticmethod
  def _dohash(s):
    h = hash()
    h.update(s.encode())
    return h.hexdigest()

  def checkpass(self,username,password):
    password = LogonDB._dohash(password)
    c = self.data.conn.cursor()
    c.execute("SELECT count(*) FROM pwdb WHERE username = ? AND
password = ?",(username,password))
    if c.fetchone()[0]==1 :return True
    return False

  def initdb(self):
    conn=sqlite3.connect(self.db)
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS pwdb(username unique not
null,password not null);")
    c.execute('INSERT OR IGNORE INTO pwdb
VALUES("admin",?)',(LogonDB._dohash("admin"),))
    conn.commit()
    conn.close()
    self.data=threading.local()
```

```
    def connect(self):
      '''call once for every thread as sqlite connection objects cannot
be shared among threads.'''
      self.data.conn = sqlite3.connect(self.db)
```

The definition of the database consists of a single table `pwdb` that is defined in the highlighted line (and only if that table does not yet exist). The `pwdb` table consists of two columns, namely, `username` and `password`. By marking both columns as `not null`, we ensure that we cannot enter empty values in any of them. The `username` column is also marked as `unique` because a username may only occur once. This database schema of a single table can be depicted in the following diagram where each column has a header with a name and several lines that list the attributes of a certain column (as our database design gets more elaborate, we will rely more on these diagrams and less on a detailed expose of the SQL code):

Pwdb	
Username	**password**
not null	not null
unique	

Anyone familiar with other dialects of SQL might have noticed that the column definitions lack any kind of type. This is deliberate: SQLite allows us to store any kind of value in a column, just as Python allows us to store any kind of value in a variable. The type of the value is directly associated with the value, not with the column or variable. SQLite does support the notion of affinity or preferred type and we will encounter that in other tables we will create in this book.

Besides creating a table, if needed (in the `initdb()` method, highlighted), we also initialize it with a username/password combination of admin/admin if the admin username is not yet there. If it is, we leave it as it is because we do not want to reset an altered password, but we do want to make sure that there is an admin username present. This is accomplished by the `insert or ignore` statement because the `insert` of an `admin` username into a table that already contains one would fail because of the unique constraint. Adding the non standard or `ignore` clause will ignore such an occurrence, in other words, it will not insert a new record with a username of admin if it is already there.

The `insert` statement also illustrates that we store passwords not as plaintext, but as hashed values (that are extremely hard to convert back to plaintext again). The hash method we use here is SHA1 and is imported as `hash()` from Python's `hashlib` module. The conversion from plaintext is handled by the `_dohash()` static method (marked as private by leading underscore in its name but note that in Python, this is a convention only, as there really aren't any private methods).

The way we store passwords in this example is still not safe enough for production environments, but implementing a more secure solution is out of scope for this book. I strongly suggest reading `http://www.aspheute.com/english/20040105.asp` for more on this subject.

The `initdb()` method also takes care of creating an object that can be used to store data that is local to a thread with the `threading.local()` function. Because, normally, all data in threads is shared, we have to use this function to create a place to store a database connection object that is different for each thread. If we were to store such a connection object in a global variable, each thread would have access to the same database connection and this is not allowed in SQLite.

The fact that we store passwords as hashed values implies that checking username/password combinations necessarily involves converting a plaintext password as well before it can be checked for existence. This is implemented in the `checkpass()` method (highlighted). The password argument is converted with the `_dohash()` method before being passed to the `execute()` method.

The SQL statement itself then counts the number of rows in the `pwdb` table that contain the given username and (hashed) password and retrieves the result. The result is a single row containing a single value, the number of matching rows. If this is one, we have a valid username/password combination, otherwise we don't. We do not discriminate between the cases where the username is unknown or whether there is more than a single row containing the same username. This is because the latter situation is unlikely to happen because of the `unique` constraint on the username column.

Have a go hero – adding new username/passwords

Our `LogonDB` class does not yet have a method to add a new username/password combination to the database. How would you implement one?

Hint: You need to provide both an exposed method that offers a page with a form where one can enter a new username and password and a method that may act as an action attribute in a `<form>` element and that is passed the username and password as parameters.

Note that this method has to check not only that the user is authenticated but also that the user that adds the new username/password is the admin, otherwise everyone could add new accounts! A sample implementation is already provided in `logondb.py`.

Tasklist II – storing tasks in a database

Now that we have seen how we may use a database engine to store persistent data and how to access this data from a CherryPy application, let's apply this new knowledge to the tasklist application we designed in the previous chapter. Of course, there is more to an application than storing data and we will also revamp the user interface in order to make it more responsive and slightly simpler to maintain.

Improving interactivity with AJAX

When you look at the difference between applications that are standalone on a PC versus a web application, you might notice few differences at a first glance. However, if you look more closely, there is a major difference: In the standalone application when something changes in the display, only those onscreen elements are redrawn that are actually modified.

In traditional web pages, this is completely different. Clicking a button that changes the sort order of a list, for example, might not only retrieve and redraw that list again, but would retrieve a complete page, including all side bars, navigation headers, advertisements, and what not.

If that unmodified content is slow to retrieve over the internet, the whole web page might feel sluggish, even more so if the whole webpage is waiting for the arrival of the last piece of information to display itself in its full glory. When web pages evolved to mimic applications, this difference in the interactive experience quickly became a nuisance and people started thinking about solutions.

One of the most prominent of those solutions is AJAX. It's an abbreviation for asynchronous JavaScript and XML, that is, a method to retrieve data by using the browser's built-in JavaScript capabilities. Nowadays, every browser supports AJAX and the jQuery library smoothes out most browser inconsistencies. The XML part in the name is no longer relevant as the data that might be retrieved with an AJAX call might be just about anything: besides XML and its close cousin HTML, **JavaScript Object Notation** (**JSON)** is a popular format to transmit data that might be processed even more simply than XML by the JavaScript interpreter in the browser.

The asynchronous bit in the AJAX name is still relevant, however: most AJAX calls that retrieve data return immediately without waiting for the result. However, they do call a function when the data retrieval is complete. This ensures that other parts of the application are not stalled and that the overall interactive experience of the web application can be improved.

Time for action – getting the time with AJAX

Enter the following code and run it. If you point your web browser to the familiar `http://localhost:8080` address, you will see something similar to the picture below with the time changing every five seconds or so. (The code is also available as `timer.py`)

What just happened?

Our small CherryPy application offers just two methods (both highlighted in the code). The `index()` method returns a minimalistic HTML page with some static text and a small piece of JavaScript that takes care of retrieving the current time from the server. It also features a `time()` method that simply returns the current time as plain text.

Chapter4/timer.py

```
import cherrypy
import os.path
from time import asctime

current_dir = os.path.dirname(os.path.abspath(__file__))

class Root(object):

  @cherrypy.expose
  def index(self):
    return '''<html>
    <head><script type="text/javascript" src="/jquery.js" ></script></
head>
    <body><h1>The current time is ...</h1><div id="time"></div>
    <script type="text/javascript">
    window.setInterval(function(){$.ajax({url:"time",cache:false,succe
ss:function(data,status,request){
      $("#time").html(data);
    }});},5000);
    </script>
    </body>
    </html>'''

  @cherrypy.expose
  def time(self,_=None):
    return asctime()

cherrypy.quickstart(Root(),config={
```

```
        '/jquery.js':
        { 'tools.staticfile.on':True,
        'tools.staticfile.filename':os.path.join(current_
    dir,"static","jquery","jquery-1.4.2.js")
        }
      })
```

The magic is in that small piece of JavaScript (highlighted). This script is executed once the static page is loaded and it calls the `setInterval()` method of the `window` object. The arguments to the `setInterval()` method are an anonymous function and a time interval in milliseconds. We set the time interval to five seconds. The function passed to `setInterval()` is called at the end of each interval.

In this example, we pass an anonymous function to `setInterval()` that relies on jQuery's `ajax()` function to retrieve the time. The `ajax()` function's only argument is an object that may contain numerous options. The `url` option tells which URL to use to retrieve the data from, in this case, the relative URL `time` (relative to the page that serves the content the script is embedded in, `/`, so it actually refers to `http://localhost:8080/time`).

The `cache` option is set to `false` to prevent the browser from using a cached result when instructed to get the time URL it has seen already. This is ensured by the underlying JavaScript library by appending an extra `_` parameter (that is the name of this parameter which consists of a single underscore) to the URL. This extra parameter contains a random number, so the browser will regard each call as a call to a new URL. The `time()` method is defined to accept this parameter because otherwise CherryPy would raise an exception, but the contents of the parameter are ignored.

The `success` option is set to a function that will be called when the data is successfully received. This function will receive three arguments when called: the data that was retrieved by the `ajax()` function, the status, and the original request object. We will only use the data here.

We select the `<div>` element with the `time` ID and replace its contents by passing the data to its `html()` method. Note that even though the `time()` method just produces text, it could just as easily have returned text containing some markup this way.

We explicitly instructed the `ajax()` function not to cache the result of the query, but instead we could also decorate our `time()` method with CherryPy's `expires` tool. This would instruct the `time()` method to insert the correct http headers in response to instruct the browser not to cache the results. This is illustrated in the following code (available in `timer2.py`):

```
@cherrypy.tools.expires(secs=0,force=True)
  @cherrypy.expose
  def time(self,_=None):
    return asctime()
```

Using the `@cherrypy.tools.expires` decorator means we do not have to instruct the `ajax()` method not to cache the result, which gives us the option to use a shortcut method. The JavaScript code may then be rewritten to use jQuery's `load()` method, shown as follows:

```
<script type="text/javascript">
window.setInterval(function(){$("#time").load("time");},5000);
</script>
```

The `load()` method is passed the URL where it will retrieve the data and, upon success, will replace the contents of the selected `DOMelement` with the data it received.

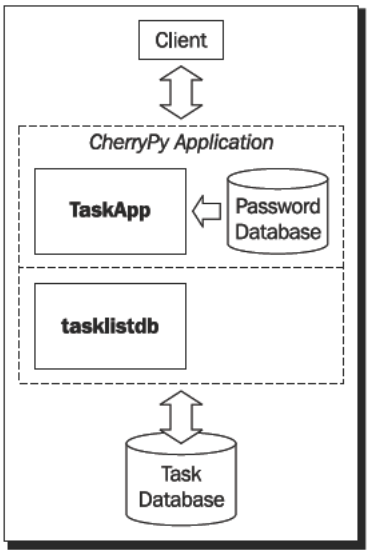
jQuery provides many AJAX shortcut methods and all these methods share a common set of defaults that may be set using the `ajaxSetup()` function. For example, to make sure all AJAX methods will not cache any returned result, we could call it like this: `$.ajaxSetup({cache:false});`

Redesigning the Tasklist application

The tasklist application will consist of two parts: an authentication part for which we will reuse the `LogonDB` class and new `TaskApp` class. The `TaskApp` class will implement the methods necessary to deliver the page with an overview of all tasks for the authenticated user plus additional methods to respond to AJAX requests.

Instead of a filesystem, SQLite will be used to store the tasks for all users. Note that this is a separate database from the one used to store usernames and passwords. Such a setup allows us to keep the authentication functionality separate from other concerns, allowing for easier reuse. Once the user is authenticated, we do, of course, use his/her username to identify the tasks belonging to him/her.

Access to the task database will be encapsulated in a `tasklistdb` module. It provides classes and methods to retrieve, add, and modify tasks for a given user. It is not concerned with checking access permission, as this is the responsibility of the `TaskApp` class. You can picture this separation as a two layer model, the top layer checking user credentials and serving content, and the bottom layer actually interfacing with a database.



Database design

The design of our task database (the database *schema*) is very straightforward. It consists of a single table, which contains columns to define a task.

Task				
task_id	**description**	**duedate**	**completed**	**user_id**
integer				
primary key				
autoincrement				

Most columns do not have a specific type defined, as SQLite will let us store anything in a column. Furthermore, most columns do not have special constraints except for the `task_id` column that we designate to be the primary key. We do explicitly type the `task_id` column as an integer and designate it as `autoincrement`. This way, we do not have to set the value of this column explicitly, but a new unique integer will be inserted for us every time we add a new task to the table.

Time for action – creating the task database

First, let us take some time to familiarize with the steps necessary to create a new database from Python.

Enter the following code and run it (It is also available as `taskdb1.py`).

Chapter4/taskdb1.py

```
import sqlite3

database=':memory:'

connection = sqlite3.connect(database)

cursor=connection.executescript('''
create table if not exists task (
  task_id integer primary key autoincrement,
  description,
  duedate,
  completed,
  user_id
);
''')
connection.commit()

sql = '''insert into task (description,duedate,completed,user_id)
values(?,?,?,?)'''
cursor.execute(sql,('work'          ,'2010-01-01',None,'alice'))
cursor.execute(sql,('more work'     ,'2010-02-01',None,'alice'))
cursor.execute(sql,('work'          ,'2010-03-01',None,'john'))
cursor.execute(sql,('even more work','2010-04-01',None,'john'))

connection.commit()

connection.close()
```

It will create a temporary database in memory and defines a task table. It also populates this table with a number of tasks using INSERT statements.

What just happened?

After establishing a connection to the database, the first task is to create the `task` table (highlighted). Here we use the `executescript()` method of the `connection` object, because this method allows us to pass more than one SQL statement in one go. Here our database schema consists of a single `create` statement so the `execute()` method would do just as well, but normally when creating a database, we create a number of tables and then passing all necessary SQL statements together is very convenient.

When you look at the `create` statement, you may notice it features a `if not exists` clause. This is completely redundant in this example because a freshly opened in-memory database is always empty, but should our database reside on disk, it might contain all the tables we want already. Once we have created the table, we commit our changes to the database with the `commit()` method.

The second highlighted line shows how we may create an insert statement that will insert new records in the task table. The values we will insert are placeholders, each represented by a question mark. In the next four lines, we execute this insert statement and supply a tuple of values that will be inserted in place of those placeholders.

Time for action – retrieving information with select statements

In SQL, the `select` statement can be used to retrieve records from a database. How would you express a query to retrieve all tasks belonging to the user john?

Answer: `select * from task where user_id = 'john'`

We may implement this in Python as follows (only relevant lines shown, complete implementation is available as `taskdb2.py`):

Chapter4/tasktdb2.py

```
connection.row_factory = sqlite3.Row
sql = """select * from task where user_id = 'john'"""
cursor.execute(sql)
tasks = cursor.fetchall()
for t in tasks:
  print(t['duedate'],t['description'])
```

What just happened?

The first line in the code is normally placed just after establishing a connection to the database and ensures that any row returned from a `fetchone()` or `fetchall()` method are not plain tuples, but `sqlite3.Row` objects. These objects behave just like tuples, but their fields can be indexed by the name of the column they represent as well.

The query is executed by passing it to the `execute()` method of the cursor attribute (highlighted) and the results are then retrieved with the `fetchall()` method that will return a list of tuples, each tuple representing a matching record, its elements equal to the columns. We print some of those elements by indexing the tuples with the column names we are interested in.

When `taskdb2.py` is run, the output will show a list of task records, each with a date and a description:

```
C:\Tasklist II>python taskdb2.py
2010-03-01 work
2010-04-01 even more work
```

Pop quiz – using variable selection criteria

Most of the time we would like to pass the `user_id` to match as a variable. As we saw in the insert statements used in `taskdb1.py`, it is possible to construct a query using `?` as placeholders. This way, we could pass a variable containing a `user_id` to the execute method. How would you refactor the code to select all records for a user whose `user_id` is contained in the variable `username`?

TaskDB – interfacing with the database

Now we are ready to take on the real implementation of the database interface needed for the tasklist application.

The database interface layer will have to provide functionality to initialize a database and to provide thread-safe ways to create, retrieve, update, and delete tasks (collectively, often called **CRUD**) as well as list all tasks for a given user. The code to do this is contained in two classes, `Task` and `TaskDB` (both available in `tasklistdb.py`). `TaskDB` encapsulates the connection to the database and contains code to initialize the database as well as methods to retrieve a selection of tasks and to create new tasks. These tasks are implemented as instances of the `Task` class and a `Task` instance may be updated or deleted.

Time for action – connecting to the database

Let's first have a look at the `TaskDB` class. It consists of a constructor `__init__()` that takes the filename where the database will reside as a parameter. It calls a private method to initialize this database, and like the `LogonDB` class, creates some storage to hold connection objects for each thread (highlighted). It also defines a `connect()` method that should be called once for each thread and stores a thread-specific connection object. It also sets the `row_factory` attribute of the connection to `sqlite3.Row`. This causes the tuples returned by, for example, `fetchall()` to have their fields named after the columns they represent. This makes sense as `t['user_id']` is a lot more self documenting than `t[1]`, for example.

Chapter4/tasklistdb.py

```
class TaskDB:

    def __init__(self,db):
      self.data = threading.local()
      self.db = db
      self._initdb()

    def connect(self):
      '''call once for every thread'''
      self.data.conn = sqlite3.connect(self.db)
      self.data.conn.row_factory = sqlite3.Row
```

What just happened?

The code for the `__init__()` method did not initialize any table in the database itself, but delegated this to the `_initdb()` method. This method starts with an underscore so it is private by convention (but by convention only). It is meant to be called just from `__init__()` and initializes the database, if necessary. It opens a connection to the database and executes a multiline statement (highlighted). Here we use `create if not exists` to create the `task` table, but only if it is not already present. So if we start the application for the first time, the database will be completely empty and this statement will create a new table named task. If we start the application again later, this statement will not do anything. Before closing the connection, we commit our changes.

Chapter4/tasklistdb.py

```
def _initdb(self):
    '''call once to initialize the metabase tables'''
    conn = sqlite3.connect(self.db)
    conn.cursor().executescript('''
    create table if not exists task (
      task_id integer primary key autoincrement,
      description,
      duedate,
      completed,
      user_id
    );
    '''
    )
    conn.commit()
    conn.close()
```

Time for action – storing and retrieving information

The final part of the `TaskDB` class defines three methods, `create()` that will create a completely new `Task` instance, `retrieve()` that will fetch a task from the `task` table given a `task_id` and return it as a `Task` instance, and `list()` that will return a list of `task_id`s for a given user.

We separated `retrieve()` and `list()` because retrieving an object complete with all its attributes might be quite expensive and not always needed. For example, if we were to select a list with thousands of tasks, we would likely display them as a page of about twenty tasks each. If we were to retrieve complete information for all those tasks, we might have to wait a while, so we might choose to instantiate only a first page-full of them and fetch the rest on an as-needed basis as the users step through the pages. We will encounter this pattern a few more times in this book.

The `create()` method itself simply passes on all parameters to the `Task` constructor together with the thread local storage that holds the database connection. It returns the resulting `Task` instance.

The `retrieve()` method takes the username and the ID of the task to retrieve. The username is taken as a sanity check, but not strictly necessary. If a record is found that matches both the `task_id` and the `username`, a `Task` instance is created and returned (highlighted). If no such record could be found, a `KeyError` exception is raised.

The `list()` method returns a list of `task_id`s for a given user. It constructs this list from the list of tuples returned by taking the first (and only) item from each tuple (highlighted).

Chapter4/tasklistdb.py

```
  def create (self, user=None, id=None, description='', duedate=None,
completed=None):
    return Task(self.data, user=user, id=id, description=description,
duedate=duedate, completed=completed)
  def retrieve(self, user,id):
    sql = """select * from task where task_id = ? and user_id = ?"""
    cursor = self.data.conn.cursor()
    cursor.execute(sql,(id,user))
    tasks = cursor.fetchall()
    if len(tasks):
      return self.create(user, tasks[0]['task_id'], tasks[0]
['description'], tasks[0]['duedate'], tasks[0]['completed'])
    raise KeyError('no such task')
  def list(self,user):
    sql = '''select task_id from task where user_id = ?'''
    cursor = self.data.conn.cursor()
    cursor.execute(sql,(user,))
    return [row[0] for row in cursor.fetchall()]
```

The constructor for `Task` takes a number of optional parameters together with a mandatory username and a `taskdb` parameter that point to the thread local data that holds the database connections. If the `duedate` parameter is not given, it assigns it the date of today (highlighted).

What just happened?

The construction of Task instances in the previous code deserves a closer look. Based on the value of the `id` parameter, the constructor can do two things.

If the `id` is known, this `Task` instance is constructed based on data just retrieved from a database query so there is nothing more to be done as all parameters are already stored as instance variables.

However, if `id` is not given (or `None`), we apparently are creating a completely new `Task` that is not already present in the database. Therefore, we have to insert it into the task table using an `insert` statement (highlighted).

We do not pass a new `task_id` as a value to this `insert` statement, but one will be created for us because we defined the `task_id` column as `integer primary key autoincrement`. This generated number is available from the cursor's `lastrowid` attribute and we store that for later reuse. All this is quite SQLite-specific, for more information, refer to the information box.

Only an `integer primary key` column can be defined as `autoincrement` and only an `integer primary key autoincrement` column will mapped to the internal `rowid` column (and that is not even a real column). All this is very useful, but also quite SQLite-specific. More information on this subject can be found on the SQLite FAQ at `http://www.sqlite.org/faq.html` and in the section on rowid in the SQL reference at `http://www.sqlite.org/lang_createtable.html#rowid`.

Chapter4/tasklistdb.py

```
class Task:
  def __init__(self,taskdb,user,id=None,description='',duedate=None,co
mpleted=None):
    self.taskdb=taskdb
    self.user=user
    self.id=id
    self.description=description
    self.completed=completed
    self.duedate=duedate if duedate != None else date.today().
isoformat()
    if id == None:
      cursor = self.taskdb.conn.cursor()
      sql = '''insert into task (description,duedate,completed,user_
id) values(?,?,?,?)'''
      cursor.execute(sql,(self.description,self.duedate,self.
completed,self.user))
      self.id = cursor.lastrowid
      self.taskdb.conn.commit()
```

Time for action – updating and deleting information

Updating the record for a `Task` is all about constructing the correct `update` query. `update` will alter any records that match the conditions in the `where` clause. It will change only those columns mentioned in its `set` clause so we start by constructing this `set` clause (highlighted).

Joining a list of parameters and interpolating it into an SQL query might be a bit overdone but if we later want to add an extra attribute, this would be very simple (and our SQL query string now fits on a single line, making it a lot easier to read and typeset).

Once we have executed the insert, we check the number of rows affected. This value is available as the `rowcount` attribute of the `cursor` object and should be `1` as we used the unique `task_id` to select the records. If it isn't `1`, something strange has happened and we roll back the insert and raise an exception. If it went well, we commit our changes.

Chapter4/tasklistdb.py

```
    def update(self,user):
      params= []
      params.append('description = ?')
      params.append('duedate = ?')
      params.append('completed = ?')
      sql = '''update task set %s where task_id = ? and user_id = ?'''
      sql = sql%(",".join(params))
      conn = self.taskdb.conn
      cursor = conn.cursor()
      cursor.execute(sql, (self.description,self.duedate,self.
completed,self.id,user))
      if cursor.rowcount != 1 :
        debug('updated',cursor.rowcount)
        debug(sql)
        conn.rollback()
        raise DatabaseError('update failed')
      conn.commit()
```

To delete a task with a given task ID, all we have to do is execute a `delete query` on the `task` table with an expression in the `where` clause that matches our `task_id`, just like we did for an update. We do check that our delete query affects a single record only (highlighted) and roll back otherwise. This shouldn't happen, but it is better to be safe than sorry.

```
    def delete(self,user):
      sql = '''delete from task where task_id = ? and user_id = ?'''
      conn = self.taskdb.conn
      cursor = conn.cursor()
      cursor.execute(sql,(self.id,user))
      if cursor.rowcount != 1:
        conn.rollback()
        raise DatabaseError('no such task')
      conn.commit()
```

Testing

Developing software without testing it is a little bit like driving a car with your eyes closed: if the road is straight you might get surprisingly far, but chances are you will crash within a few seconds. Testing, in other words, is good.

It does take time, however, to test an application thoroughly, so it makes sense to automate the testing process as much as possible. If tests can be executed easily, it encourages developers to run these tests often. This is desirable when the implementation changes. It can also act as a sanity check just before a new release. So although writing serious tests may sometimes take about as long as writing the code itself, this is a solid investment, as it might prevent many unwelcome surprises if the code is changed or the environment in which the code is deployed is altered.

There are many aspects of an application that you might like to test, but not all lend themselves to automatic testing, like user interaction (although tools like Selenium can get you quite far. More information on this tool is available at `http://seleniumhq.org/`). However, other parts are quite simple to automate.

Python comes with a `unittest` module that simplifies the task of repeatedly testing small functional units of code. The idea of unit testing is to isolate small chunks of code and define its expected behavior by asserting any number of expectations. If one of those assertions fails, the test fails. (There is much more to unit testing than can be fully covered in this book. Here we cover just the bare minimum to get a taste of the possibilities and we cover a few examples that are intended to give you enough information to understand the test suites supplied with the example code for this book. If you would like to read more on unit testing in Python, a good starting point would be *Python Testing* by *Daniel Arbuckle*, *Packt Publishing*, 978-1-847198-84-6).

Python's `unittest` module contains a number of classes and functions that enable us to write and run groups of tests and their associated assertions. For example, say we have a module called `factorial` that defines a function `fac()` to calculate a factorial.

A factorial of a number n is the product of all numbers from 1 to n inclusive. For example, `fac(4) = 4 * 3 * 2 * 1 = 24`. Zero is an exceptional case as the factorial of 0 = 1. Factorials are only defined for integers >= 0, so we design our code to raise `ValueError` exceptions if the argument `n` is not an `int` or is negative (highlighted). The factorial itself is calculated recursively. If `n` is either zero or one, we return one, otherwise we return the factorial of `n` minus one times `n`:

Chapter4/factorial.py

```
def fac(n):
  if n < 0 : raise ValueError("argument is negative")
  if type(n) != int : raise ValueError("argument is not an integer")
  if n == 0 : return 1
```

```
    if n == 1 : return 1
    return n*fac(n-1)
```

The code is available as `factorial.py`.

Time for action – testing factorial.py

The test suite to accompany `factorial.py` is called `test_factorial.py`. Run it and you should see output similar to this:

```
python test_factorial.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Three tests were executed and apparently everything went ok.

What just happened?

The code in `test_factorial.py` starts by importing both the module we want to test (`factorial`) and the `unittest` module. Then we define a single class named `Test` (highlighted) derived from `unittest.TestCase`. By deriving from this class, our class will be distinguishable as a test case to the test runner and will provide us with a number of **assertion** methods.

Our `Test` class may consist of any number of methods. The ones with names starting with `test_` will be recognized as tests by the test runner. Because the names of failing tests will be printed, it is useful to give these tests sensible names reflecting their purpose. Here we define three such methods: `test_number()`, `test_zero()`, and `test_illegal()`.

Chapter4/test_factorial.py

```
import unittest
from factorial import fac

class Test(unittest.TestCase):

  def test_number(self):
    self.assertEqual(24,fac(4))
    self.assertEqual(120,fac(5))
    self.assertEqual(720,fac(6))

  def test_zero(self):
    self.assertEqual(1,fac(0))

  def test_illegal(self):
    with self.assertRaises(ValueError):
```

```
        fac(-4)
      with self.assertRaises(ValueError):
        fac(3.1415)

if __name__ == '__main__':
  unittest.main()
```

`test_number()` tests a number of regular cases to see if our function returns something reasonable. In this case, we check three different numbers and use the `assertEquals()` method inherited from the `TestCase` class to check that the value calculated (passed as the second argument) equals the expected value (the first argument).

`test_zero()` asserts that the special case of zero indeed returns 1. It again uses the `assertEqual()` method to check whether the expected value (1) matches the value returned.

`test_illegal()` finally asserts that only positive arguments are accepted (or rather it asserts that negative values correctly raise a `ValueError` exception) and that arguments to `fac()` should be `int` or raise a `ValueError` as well.

It utilizes the method `assertRaises()` provided by `TestCase`. `assertRaises()` will return an object that can be used as a context manager in a with statement. Effectively, it will catch any exception and check whether it is an expected one. If not, it will flag the test as failed.

These methods show a familiar pattern in unit testing: a fairly small number of tests check whether the unit behaves correctly in normal cases, while the bulk of the tests are often devoted to special cases (often referred to as edge cases). And, just as important, serious effort is spent on testing that illegal cases are correctly flagged as such.

The last thing we find in `test_factorial.py` is a call to `unittest.main()`, the test runner. It will look for any defined classes deriving from `TestCase` and run any method that starts with `test_`, tallying the results.

Now what have we gained?

If we would change, for example, the implementation of `fac()` to something that does not use recursion like the following code, we could rapidly check that it behaves as expected by running `test_factorial.py` again.

```
from functools import reduce
def fac(n):
  if n < 0 : raise ValueError("factorial of a negative number is not
defined")
  if type(n) != int : raise ValueError("argument is not an integer")
  if n == 0 : return 1
  if n == 1 : return 1
  return reduce(lambda x,y:x*y,range(3,n+1))
```

The special case handling remains the same, but the highlighted line shows that we now calculate the factorial with Python's `reduce()` function from the `functools` module. The `reduce()` function will apply a function to the first pair of items in a list and then again to the result of this and each remaining item. The product of all numbers in a list can be calculated by passing `reduce()` a function that will return the product of two arguments, in this case, our lambda function.

More on the `reduce()` function can be found in the documentation of the `functools` module, Python's powerful functional programming library: `http://docs.python.org/py3k/library/functools.html`.

Pop quiz – spotting the error

1. Can you anticipate any errors in the previous code? Which test method will fail?
   - `test_number()`
   - `test_zero()`
   - `test_illegal()`

Time for action – writing unit tests for tasklistdb.py

Run `test_tasklistdb.py` (provided in the code distribution for this chapter). The output should be a list of test results:

```
python test_tasklistdb.py
......
----------------------------------------------------------------------
Ran 6 tests in 1.312s

OK
```

What just happened?

Let us look at one of the classes defined in `test_tasklistdb.py`, `DBentityTest`. `DBentityTest` contains a number of methods starting with `test_`. These are the actual tests and they verify whether some common operations like retrieving or deleting tasks behave as expected.

Chapter4/test_tasklistdb.py

```
from tasklistdb import TaskDB, Task, AuthenticationError,
DatabaseError
import unittest
from os import unlink,close
from tempfile import mkstemp
```

```
(fileno,database) = mkstemp()
close(fileno)

class DBentityTest(unittest.TestCase):

  def setUp(self):
    try:
      unlink(database)
    except:
      pass
    self.t=TaskDB(database)
    self.t.connect()
    self.description='testtask'
    self.task = self.t.create(user='testuser',description=self.
description)

  def tearDown(self):
    self.t.close()
    try:
      unlink(database)
    except:
      pass

  def test_retrieve(self):
    task = self.t.retrieve('testuser',self.task.id)
    self.assertEqual(task.id,self.task.id)
    self.assertEqual(task.description,self.task.description)
    self.assertEqual(task.user,self.task.user)

  def test_list(self):
    ids = self.t.list('testuser')
    self.assertListEqual(ids,[self.task.id])

  def test_update(self):
    newdescription='updated description'     self.task.
description=newdescription
    self.task.update('testuser')
    task = self.t.retrieve('testuser',self.task.id)
    self.assertEqual(task.id,self.task.id)
    self.assertEqual(task.duedate,self.task.duedate)
    self.assertEqual(task.completed,self.task.completed)
    self.assertEqual(task.description,newdescription)

  def test_delete(self):
    task = self.t.create('testuser',description='second task')
    ids = self.t.list('testuser')
    self.assertListEqual(sorted(ids),sorted([self.task.id,task.id]))
    task.delete('testuser')
```

```
        ids = self.t.list('testuser')
        self.assertListEqual(sorted(ids),sorted([self.task.id]))
        with self.assertRaises(DatabaseError):
          task = self.t.create('testuser',id='short')
          task.delete('testuser')
if __name__ == '__main__':
  unittest.main(exit=False)
```

All these `test_` methods depend on an initialized database containing at least one task and an open connection to this database. Instead of repeating this setup for each test, `DBentityTest` contains the special method `setUp()` (highlighted) that removes any test database lingering around from a previous test and then instantiates a `TestDB` object. This will initialize the database with proper table definitions. Then it connects to this new database and creates a single task object. All tests now can rely upon their initial environment to be the same. The corresponding `tearDown()` method is provided to close the database connection and remove the database file.

The file that is used to store the temporary database is created with the `mkstemp()` function from Python's `tempfile` module and stored in the global variable database. (`mkstemp()` returns the number of the file handle of the opened as well, which is immediately used to close the file as we are only interested in the name of the file.)

The `test_list()` and `test_delete()` methods feature a new assertion: `assertListEqual()`. This assertion checks whether two lists have the same items (and in the same order, hence the `sorted()` calls). The `unittest` module contains a whole host of specialized assertions that can be applied for specific comparisons. Check Python's online documentation for the `unittest` module for more details (`http://docs.python.org/py3k/library/unittest.html`).

Many of the modules we develop in this book come bundled with a suite of unit tests. We will not examine those tests in any detail, but it might be educational to check some of them. You should certainly use them if you experiment with the code as that is exactly what they are for.

Designing for AJAX

Using AJAX to retrieve data not only has the potential to make the tasklist application more responsive, but it will also make it simpler. This is achieved because the HTML will be simpler as there will be no need for the many `<form>` elements we created to accommodate the various delete and done buttons. Instead, we will simply act on click events bound to buttons and call small methods in our CherryPy application. All these functions have to do is perform the action and return ok, whereas in the previous version of our application, we would have to return a completely new page.

In fact, apart from a number of `<script>` elements in the `<head>`, the core HTML in the body is rather short (the `<header>` element and the extra elements in the `<div>` element with a `taskheader` class are omitted for brevity):

```
<body id="itemlist">
  <div id="content">
    <div class="header"></div>
    <div class="taskheader"></div>
    <div id="items"></div>
    <div class="item newitem">
      <input type="text" class="duedate left editable-date tooltip"
           name="duedate" title="click for a date" />
      <input type="text" class="description middle tooltip"
           title="click to enter a description" name="description"/>
      <button type="submit" class="add-button"
            name="add" value="Add" >Add</button>
    </div>
  </div>
</body>
```

The `<div>` element containing the input fields and a submit button takes up most of the space. It structures the elements that make up the line that allows the user to add new tasks. The `<div>` element with the ID `items` will hold a list of tasks and will be initialized and managed by the JavaScript code using AJAX calls.

The JavaScript code in `tasklistajax.js` serves a number of goals:

- Initializing the list of items
- Styling and enhancing UI elements with interactive widgets (like a `datepicker`)
- Maintaining and refreshing the list of tasks based on button clicks

Let's have a look at `tasklistajax.js`.

Chapter4/static/js/tasklistajax.js

```
$.ajaxSetup({cache:false});$.ajaxSetup({cache:false});

function itemmakeup(data,status,req){
    $(".done-button").button(  {icons: {primary: 'ui-icon-check'
}, text:false});
    $(".del-button").button(  {icons: {primary: 'ui-icon-trash'     },
text:false});
    $("#items input.duedate").sort(
      function(a,b){return $(a).val() > $(b).val() ? 1 : -1;},
      function(){ return this.parentNode; }).addClass("just-sorted");
    // disable input fields and done button on items that are already
marked as completed
```

```
      $(".done .done-button").button( "option", "disabled", true );
      $(".done input").attr("disabled","disabled");
      $( "#items .editable-date" ).datepicker({
        dateFormat: $.datepicker.ISO_8601,
        onClose: function(dateText,datePicker){ if(dateText != '')
{$(this).removeClass("inline-label");}}
      });
    };

$(document).ready(function(){
  $(".header").addClass("ui-widget ui-widget-header");
  $(".add-button").button(   {icons: {primary: 'ui-icon-plusthick' },
text:false}).click(function(){
        $(".inline-label").each(function() {
      if($(this).val() === $(this).attr('title')) {
        $(this).val('');
      };
        })
    var dd=$(this).siblings(".duedate").val();
    var ds=$(this).siblings(".description").val();
    $.get("add",{description:ds, duedate:dd},function(data,status,req)
{
      $("#items").load("list",itemmakeup);
    });
        return false; // prevent the normal action of the button click
    });
    $(".logoff-button").button({icons: {primary: 'ui-icon-
closethick'}, text:false}).click(function(){
    location.href = $(this).val();
    return false;
  });
  $(".login-button").button( {icons: {primary: 'ui-icon-play'      },
text:false});
  $(":text").addClass("textinput");
  $(":password").addClass("textinput");
  $( ".editable-date" ).datepicker({
    dateFormat: $.datepicker.ISO_8601,
    onClose: function(dateText,datePicker){ if(dateText != '')
{$(this).removeClass("inline-label");}}
  });
  // give username field focus (only if it's there)
  $("#username").focus();

  $(".newitem input").addClass("ui-state-highlight");
  $(".done-button").live("click",function(){
```

```
        var item=$(this).siblings("[name='id']").val();
        var done=$(this).siblings(".completed").val();
        $.get("done",{id:item, completed:done},function(data,status,req)
{
          $("#items").load("list",itemmakeup);
        });
        return false;
      });
    $(".del-button").live("click",function(){
        var item=$(this).siblings("[name='id']").val();
        $.get("delete",{id:item},function(data,status,req){
          $("#items").load("list",itemmakeup);
        });
        return false;
      });
    $("#items").load("list",itemmakeup); // get the individual task
items
});
```

The first line establishes the defaults for all AJAX calls that we will use. It makes sure that the browser will not cache any results.

Initializing the list of items once the page is loaded is done in the final highlighted line of code. It calls the `load()` method with a URL that will be handled by our application and will return a list of tasks. If the call to `load()` is successful, it will not only insert this data in the selected `<div>` element, but also call the function `itemmakeup()` passed to it as a second argument. That function, `itemmakeup()`, is defined in the beginning of the file. It will style any `<button>` element with a `done-button` or `del-button` class with a suitable icon. We do not add any event handlers to those buttons here, which is done elsewhere as we will see shortly.

Next, we use the `sort` plugin to sort the items (highlighted), that is, we select any input field with the `duedate` class that are children of the `<div>` element with the ID `items` (we do not want to consider input fields that are part of the new item div for example).

The sort plugin is available as sort.js and is based on code by James Padolsey: `http://james.padolsey.com/javascript/sorting-elements-with-jquery/`. The plugin will sort any list of HTML elements and takes two arguments. The first argument is a comparison function that will return either 1 or -1 and the second argument is a function that when given an element will return the element that should actually be moved around. This allows us to compare the values of child elements while swapping the parent elements they are contained in.

For example, here we compare the due dates. That is, the content of the selected `<input>` elements, as retrieved by their `val()` method, but we sort not the actual input fields but their parents, the `<div>` elements containing all elements that make up a task.

Finally, `itemmakeup()` makes sure any button marked with a `done` class is disabled as is any input element with that class to prevent completed tasks from being altered and changes any input element with an `editable-date` class into a datapicker widget to allow the user to choose a completion date before marking a task as done.

Click handlers

Besides styling elements, the `$(document).ready()` function adds click handlers to the add, done, and delete buttons (highlighted).

Only one add button is created when the page is created, so we can add a click handler with the `click()` method. However, new done and delete buttons may appear each time the list of items is refreshed. To ensure that freshly appearing buttons that match the same selection criteria receive the same event handler as the ones present now, we call the `live()` method.

jQuery's `live()` method will make sure any event handler is attached to any element that matches some criterion, now or in the future. More on jQuery's event methods can be found at `http://api.jquery.com/category/events/`.

Apart from the way we bind an event handler to a button, the actions associated with a click are similar for all buttons. We retrieve the data we want to pass to the server by selecting the appropriate input elements from among the button's siblings with the `siblings()` method. As each task is represented by its own `<div>` element in the list and the `<button>` and `<input>` elements are all children of that `<div>` element, so selecting sibling input elements only ensures that we refer to elements of a single task only.

To get a better understanding of what we are selecting with the `siblings()` method, take a look at some of the (simplified) HTML that is generated for the list of items:

```
<div id="items">
  <div class="item"><input name=""/> … <button name="done"></div
  <div class="item"><input name=""/> … <button name="done"></div
  …
</div>
```

So each `<div>` that represents a task contains a number of `<input>` elements and some `<button>` elements. The siblings of any `<button>` element are the elements within the same `<div>` (without the button itself).

When we have gathered the relevant data from the input elements, this data is then passed to a `get()` call. The `get()` function is another AJAX shortcut that will make an HTTP GET request to the URL given as its first argument (a different URL for each button type). The data passed to the `get()` function is appended to the GET request as parameters. Upon success, the function passed as the third argument to `get()` is called. This is the same `itemmakeup()` function that refreshes the list of items that was used when the page was first loaded.

The application

With the JavaScript to implement the interactivity and the means to access the database in place, we still have to define a class that can act as a CherryPy application. It is available as `taskapp.py` and here we show the relevant bits only (Its `index()` method is omitted because it simply delivers the HTML shown earlier).

Chapter4/taskapp.py

```
class TaskApp(object):
  def __init__(self,dbpath,logon,logoffpath):
    self.logon=logon
    self.logoffpath=logoffpath
    self.taskdb=TaskDB(dbpath)

  def connect(self):
    self.taskdb.connect()
```

The constructor for `TaskApp` stores a reference to a `LogonDB` instance in order to be able to call its `checkauth()` method in exposed methods to authenticate a user. It also stores the `logoffpath`, a URL to a page that will end the user's session. The `dbpath` argument is the filename of the file that holds the tasklist database. It is used to create an instance of `TaskDB`, used in subsequent methods to access the data (highlighted).

The `connect()` method should be called for each new CherryPy thread and simply calls the corresponding method on the `TaskDB` instance.

To service the AJAX calls of the application, `TaskApp` exposes four short methods: `list()` to generate a list of tasks, `add()` to add a new task, and `done()` and `delete()` to mark a task as done or to remove a task respectively. All take a dummy argument named `_` (a single underscore) that is ignored. It is added by the AJAX call in the browser to prevent caching of the results.

`list()` is the longer one and starts out with authenticating the user making the request (highlighted). If the user is logged in, this will yield the username. This username is then passed as an argument to the `taskdb.list()` method to retrieve a list of task IDs belonging to this user.

With each ID, a `Task` instance is created that holds all information for that task (highlighted). This information is used to construct the HTML that makes up the task as visualized on screen. Finally, all HTML of the individual tasks is joined and returned to the browser.

Chapter4/taskapp.py

```
    @cherrypy.expose
    def list(self,_=None):
      username = self.logon.checkauth()
      tasks = []
      for t in self.taskdb.list(username):
        task=self.taskdb.retrieve(username,t)
        tasks.append('''<div class="item %s">
        <input type="text" class="duedate left" name="duedate"
value="%s" readonly="readonly" />
        <input type="text" class="description middle" name="description"
value="%s" readonly="readonly" />
        <input type="text" class="completed right editable-date tooltip"
title="click to select a date, then click done" name="completed"
value="%s" />
        <input type="hidden" name="id" value="%s" />
        <button type="submit" class="done-button" name="done"
value="Done" >Done</button>
        <button type="submit" class="del-button" name="delete"
value="Del" >Del</button>
        </div>'''%('notdone' if task.completed==None else 'done',task.
duedate,task.description,task.completed,task.id))
      return '\n'.join(tasks)
```

The other methods are quite similar to each other. `add()` takes `description` and `duedate` as arguments and passes them together with the username it got after authentication of the user to the `create()` method of the `TaskDB` instance. It returns '`ok`' to indicate success. (Note that an empty string would do just as well: it's the return code that matters, but this makes it more obvious to anyone reading the code).

The `delete()` method (highlighted) has one relevant argument, `id`. This ID is used together with the username to retrieve a `Task` instance. This instance's `delete()` method is then called to remove this task from the database.

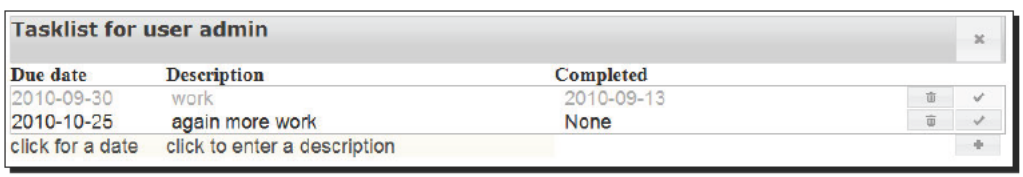
The `done()` method (highlighted) also takes an `id` argument together with `completed`. The latter either holds a date or is empty, in which case it is set to today's date. A `Task` instance is retrieved in the same manner as for the `delete()` method, but now its `completed` attribute is set with the contents of the argument of the same name and its `update()` method is called to synchronize this update with the database.

Chapter4/taskapp.py

```
    @cherrypy.expose
    def add(self,description,duedate,_=None):
      username = self.logon.checkauth()
      task=self.taskdb.create(user=username, description=description,
duedate=duedate)
      return 'ok'

    @cherrypy.expose
    def delete(self,id,_=None):
      username = self.logon.checkauth()
      task=self.taskdb.retrieve(username,id)
      task.delete(username)
      return 'ok'

    @cherrypy.expose
    def done(self,id,completed,_=None):
      username = self.logon.checkauth()
      task=self.taskdb.retrieve(username,id)
      if completed == ""  or completed == "None":
        completed = date.today().isoformat()
      task.completed=completed
      task.update(username)
      return 'ok'
```

Time for action – putting it all together

Now that we have all the requisite components in place (that is, `tasklistdb.py`, `taskapp.py`, and `tasklistajax.js`), it is straightforward to put them together. If you run the code below (available as `tasklist.py`) and point your browser at `http://localhost:8080/`, you will get a familiar looking login screen and after entering some credentials (username admin and password admin are configured by default) the resulting screen will look almost the same as the application we developed in the previous chapter, as illustrated in the following screenshot:



What just happened?

For the CherryPy application, we need a root class that can act as the root of the tree of pages we serve the user. Again, we call this class simply `Root` and assign an instance of our `TaskApp` application to the task variable and an instance of the `LogonDB` application to the logon variable (highlighted in the code below). Together with the `index()` method, this will create a tree of pages looking like this:

```
/
/logon
/task
```

If the user starts on the top-level page or on the logon page, he/she will be redirected to the `/task` page after successful authentication. Below the `/task` page are, of course, the other pages that implement the server side of the AJAX communications like, for example, `/task/add`.

Chapter4/tasklist.py

```
import cherrypy
from taskapp import TaskApp
from logondb import LogonDB
import os.path
current_dir = os.path.dirname(os.path.abspath(__file__))
theme = "smoothness"
class Root(object):
  logon = LogonDB()
  task = TaskApp(dbpath='/tmp/taskdb.db', logon=logon, logoffpath="/
logon/logoff")
  @cherrypy.expose
  def index(self):
    return Root.logon.index(returnpage='/task')
if __name__ == "__main__":
  Root.logon.initdb()
  def connect(thread_index):
    Root.task.connect()
    Root.logon.connect()
  # Tell CherryPy to call "connect" for each thread, when it starts up
  cherrypy.engine.subscribe('start_thread', connect)
  cherrypy.quickstart(Root(),config={
    '/':
    { 'log.access_file' : os.path.join(current_dir,"access.log"),
```

```
        'log.screen': False,
        'tools.sessions.on': True
        },
        '/static':
        { 'tools.staticdir.on':True,
        'tools.staticdir.dir':os.path.join(current_dir,"static")
        },
        '/jquery.js':
        { 'tools.staticfile.on':True,
        'tools.staticfile.filename':os.path.join(current_
    dir,"static","jquery","jquery-1.4.2.js")
        },
        '/jquery-ui.js':
        { 'tools.staticfile.on':True,
        'tools.staticfile.filename':os.path.join(current_
    dir,"static","jquery","jquery-ui-1.8.1.custom.min.js")
        },
        '/jquerytheme.css':
        { 'tools.staticfile.on':True,
        'tools.staticfile.filename':os.path.join(current_dir,"static","jqu
    ery","css",theme,"jquery-ui-1.8.4.custom.css")
        },
        '/images':
        { 'tools.staticdir.on':True,
        'tools.staticdir.dir':os.path.join(current_dir,"static","jquery","
    css",theme,"images")
        }
      })
```

Before the CherryPy application is started in the usual way by calling the `quickstart()` function, we first initialize the authentication database and create a function `connect()` (highlighted). This is the function we will register with CherryPy to execute each time CherryPy starts a new thread. The function will create a connection to the SQLite databases containing the authentication and tasklist data.

Have a go hero – refreshing the itemlist on a regular basis

If you were to access your tasklist from home and keep the application open and later access it from, for example, your work, any changes made to the list from work wouldn't be visible at home unless you refreshed the page manually. This is because there is nothing implemented to refresh the list of tasks regularly; it is refreshed only after some action is initiated by clicking a button.

How could you implement a regular refresh? Hint: in the first AJAX example, we encountered JavaScript's `setInterval()` method. Can you devise a way to let it replace the contents of the `<div>` element containing the list of tasks using the `load()` method?

An example implementation is available in `tasklistajax2.js`. You can either rename it to `tasklistajax.js` and run `tasklist.py` or run `tasklist2.py`.

Summary

We learned a lot in this chapter about using a database to store persistent data.

Specifically, we covered:

- The benefits of using a database engine
- How to use SQLite, a database engine distributed with Python
- How to implement a password database
- How to design and develop a database-driven tasklist application
- How to implement unit tests with Python's `unittest` module
- How to make a web application more responsive using AJAX calls

We also discussed how to make a web application respond to mouse clicks and request new data from the server without using `<form>` elements but using jQuery's `click()` and `live()` methods.

Now that we've made the first step in using a database, we're ready to create more elaborate databases designs, consisting of more than a single table, and look at the methods to define relations between these tables – which is the topic of the next chapter.

5
Entities and Relations

Most real life applications sport more than one entity and often many of these entities are related. Modeling these relations is one of the strong points of a relational database. In this chapter, we will develop an application to maintain lists of books for multiple users.

In this chapter, we will:

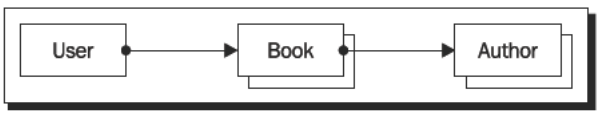
- Design and implement a data model consisting of several entities and relations
- Implement reusable entity and relation modules
- Look in depth at the necessity of clearly separating layers of functionality
- And encounter jQuery UI's autocomplete widget

So let's get started with it...

Designing a book database

Before we start to design our application, let's have a good look at the different entities that need to be handled by it. The entities we recognize are a book, an author, and a user. A book may have many attributes, but here we limit ourselves to a title, an ISBN (International Standard Book Number), and a publisher. An author has just a name, but of course, if we would like to extend that with extra attributes, like the date of birth or nationality, we can always add that later. Finally, a user is an entity with a single attribute as well, the user ID.

The next important part is to have a clear understanding of the relations between these entities. A book may be written by one or more authors, so we need to define a relation between a book entity and an author entity. Also, any number of users may own a copy of a book. This is another relation we have to define, this time, between a book entity and a user entity. The following diagram may help to see those entities and their relations more clearly:



These three entities and the relations between them need to be represented in two realms: as database tables and as Python classes. Now we could model each entity and relation separately, like we did in the previous chapter for the `tasklist` application, but all entities share a lot of common functionality so there are ample opportunities for reuse. Reuse means less code and less code equals less maintenance and often better readability. So let's see what we need to define a reusable `Entity` class.

The Entity class

From what we learned in the previous chapters, we already know there is a shared body of functionality that each class that represents an entity needs to implement:

- It needs to be able to verify that a corresponding table exists in the database and create one if it doesn't.
- It needs to implement a way to manage database connections in a thread-safe manner.

Also, each entity should present us with a **CRUD** interface:

- *Create* new object instances
- *Retrieve* individual object instances and find instances that match some criteria
- *Update* the attributes of an object instance and synchronize this data to the database
- *Delete* an object instance

That is a lot of shared functionality, but of course a book and an author are not identical: They differ in the number and type of their attributes. Before we look at the implementation, let's illustrate how we would like to use an `Entity` class to define a specific entity, for example, a car.

Time for action – using the Entity class

Let us first define for ourselves how we want to use an `Entity` class, because the interface we create must match as closely as possible the things we would like to express in our code. The following example shows what we have in mind (available as `carexample.py`):

Chapter5/carexample.py

```
from entity import Entity

class Car(Entity): pass

Car.threadinit('c:/tmp/cardatabase.db')
Car.inittable(make="",model="",licenseplate="unique")

mycar = Car(make="Volvo",model="C30",licenseplate="12-abc-3")
yourcar = Car(make="Renault",model="Twingo",licenseplate="ab-cd-12")

allcars = Car.list()

for id in allcars:
    car=Car(id=id)
    print(car.make, car.model, car.licenseplate)
```

The idea is to create a `Car` class that is a subclass of `Entity`. We therefore have to take the following steps:

1. Import the `Entity` class from the `entity` module.
2. Define the `Car` class. The body of this class is completely empty as we simply inherit all functionality from the `Entity` class. We could, of course, augment this with specific functionality, but in general, this shouldn't be necessary.
3. Initialize a connection to the database. Before we can work with the `Car` instances, an application has to initialize a connection to the database for each thread. In this example, we do not create extra threads, so there is just the single main thread of the application that needs a connection to the database. We create one here with the `threadinit()` method (highlighted).
4. Make sure an appropriate table with the necessary columns exists in the database. Therefore, we call the `inittable()` method with arguments that specify the attributes of our entity with possibly extra information on how to define them as columns in a database table. Here we define three columns: `make`, `model`, and `licenseplate`. Remember that SQLite doesn't need explicit typing, so `make` and `model` are passed as arguments with just an empty string as the value. The `licenseplate` attribute, however, is adorned with a `unique` constraint in this example.

Now we can work with `Car` instances, as illustrated in the lines that create two different objects or in the last few lines that retrieve the IDs of all `Car` records in the database and instantiate `Car` instances with those IDs to print the various attributes of a `Car`.

That is the way we would like it to work. The next step is to implement this.

What just happened?

The previous example showed how we could derive the `Car` class from `Entity` and use it. But what does that `Entity` class look like?

The definition for the `Entity` class starts off with defining a class variable `threadlocal` and a class method `threadinit()` to initialize this variable with an object that holds data that is local to each thread (the full code is available as `entity.py`).

If this `threadlocal` object does not yet have a `connection` attribute, a new connection to the database is created (highlighted) and we configure this connection by setting its `row_factory` attribute to `sqlite.Row`, as this will enable us to access columns in the results by name.

We also execute a single `pragma foreign_keys=1` statement to enable the enforcing of foreign keys. As we will see, when we discuss the implementation of relations, this is vital in maintaining a database without dangling references. This `pragma` must be set for each connection separately; therefore, we put it in the thread initialization method.

Chapter5/entity.py

```
import sqlite3 as sqlite
import threading

class Entity:
    threadlocal = threading.local()

    @classmethod
    def threadinit(cls,db):
            if not hasattr(cls.threadlocal,'connection') or
cls.threadlocal.connection is None:
                cls.threadlocal.connection=sqlite.connect(db)
                cls.threadlocal.connection.row_factory = sqlite.Row
                cls.threadlocal.connection.execute("pragma foreign_
keys=1")
            else:
                pass #print('threadinit thread has a connection
object already')
```

Next is the `inittable()` method. This should be called once to verify that the table necessary for this entity already exists or to define a table with suitable columns if it doesn't. It takes any number of keyword arguments. The names of the keywords correspond to the names of the columns and the value of such a keyword may be an empty string or a string with additional attributes for the column, for example, `unique` or an explicit type like `float`.

Although SQLite allows you to store a value of any type in a column, you may still define a type. This type (or more accurately, **affinity**) is what SQLite tries to convert a value to when it is stored in a column. If it doesn't succeed, the value is stored as is. Defining a column as float, for example, may save a lot of space. More on these affinities can be found on `http://www.sqlite.org/datatype3.html`.

Chapter5/entity.py

```
    @classmethod
    def inittable(cls,**kw):
            cls.columns=kw
            connection=cls.threadlocal.connection
            coldefs=",".join(k+' '+v for k,v in kw.items())
            sql="create table if not exists %s (%s_id integer primary
key autoincrement, %s);"%(cls.__name__,cls.__name__,coldefs)
            connection.execute(sql)
            connection.commit()
```

The column definitions are stored in the `columns` class variable for later use by the `__init__()` method and joined together to a single string. This string, together with the name of the class (available in the `__name__` attribute of a (possibly derived) class) is then used to compose a SQL statement to create a table (highlighted).

Besides the columns that we defined based on the keyword arguments, we can also create a primary key column that will be filled with a unique integer automatically. This way, we ensure that we can refer to each individual row in the table later on, for example, from a bridging table that defines a relation.

When we take our previous car example, we see that a Python statement like:

```
Car.inittable(make="",model="",licenseplate="unique")
```

Is converted to the following SQL statement:

```
create table if not exists Car (
Car_id integer primary key autoincrement,
make ,
licenseplate unique,
model
);
```

Note that the order in which we pass the keyword arguments to the `inittable()` method is not necessarily preserved as these arguments are stored in a `dict` object, and regular `dict` objects do not preserve the order of their keys.

Sometimes preserving the order of the keys in a dictionary is very desirable. In this case, column order doesn't matter much, but Python does have an `OrderedDict` class available in its `collections` module (see `http://docs.python.org/library/collections.html#collections.OrderedDict`) that we could have used. However, this would prevent us from using keywords to define each column.

Also note that there isn't any form of sanity checking implemented: anything may be passed as a value for one of the column definitions. Judging whether that is anything sensible is left to SQLite when we pass the SQL statement to the database engine with the `execute()` method.

This method will raise an `sqlite3.OperationalError` if there is a syntax error in the SQL statement. However, many issues are simply ignored. If we pass an argument like `licenseplate="foo"`, it would happily proceed, assuming `foo` to be a type it doesn't recognize, so it is simply ignored! If the execution didn't raise an exception, we finish by committing our changes to the database.

Have a go hero – checking your input

Silently ignoring things passed as arguments is not considered a good habit. Without explicit checking, a developer might not even know he/she has done something wrong, something that might backfire later.

How would you implement code to restrict the value to a limited set of directives?

Hint: Types and constraints in a SQL column definition mostly consist of single words. You could check each word against a list of allowable types, for example.

Time for action – creating instances

The next method we look at is the constructor—the `__init__()` method. It will be used to create individual instances of an entity. The constructor can be called in two ways:

- With a single `id` argument, in which case, an existing record will be retrieved from the database and the instance initialized with the column values of this record, or

- With a number of keyword arguments to create a new instance and save this as a new database record

The code to implement this behavior looks like the following:

Chapter5/entity.py

```
    def __init__(self,id=None,**kw):
        for k in kw:
            if not k in self.__class__.columns :
                raise KeyError("unknown column")
        cursor=self.threadlocal.connection.cursor()
        if id:
            if len(kw):
                raise KeyError("columns specified on
retrieval")
            sql="select * from %s where %s_id = ?"%(
            self.__class__.__name__,self.__class__.__name__)
            cursor.execute(sql,(id,))
            entities=cursor.fetchall()
            if len(entities)!=1 :
                raise ValueError("not a singular entity")
            self.id=id
            for k in self.__class__.columns:
                setattr(self,k,entities[0][k])
        else:
            cols=[]
            vals=[]
            for c,v in kw.items():
                cols.append(c)
                vals.append(v)
                setattr(self,c,v)
            cols=",".join(cols)
            nvals=",".join(["?"]*len(vals))
            sql="insert into %s (%s) values(%s)"%(
            self.__class__.__name__,cols,nvals)
            try:
                with self.threadlocal.connection as conn:
                    cursor=conn.cursor()
                    cursor.execute(sql,vals)
                    self.id=cursor.lastrowid
            except sqlite.IntegrityError:
                raise ValueError("duplicate value for unique
col")
```

The code reflects this dual use. After checking that all keywords indeed refer to the previously defined columns (highlighted), it checks whether it was passed an `id` argument. If it was, there shouldn't be any other keyword arguments. If there are additional keywords, an exception is raised. If the `id` argument is present, an SQL statement is constructed next that will retrieve the records from the associated table. Each record's primary key should match the ID.

What just happened?

Because the primary key is unique, this will match at most a single record, something that is verified after we retrieve the matching records. If we didn't fetch exactly one (1) record, an exception is raised (highlighted).

If everything went well, we initialize the attributes of the instance we are creating with the built-in `setattr()` function. The columns of the record we retrieved can be accessed by name because we initialized the `row_factory` attribute of the connection object to a `sqlite3.Row`. We also stored the names of the columns in the `columns` class variable and this lets us initialize the instance's attributes with the values of the corresponding column names (highlighted).

Creating a `Car` instance with:

```
Car(id=1)
```

Will result in a SQL statement like this:

```
select * from Car where Car_id = ?
```

Where the question mark is a placeholder for the actual value that is passed to the `execute()` method.

The second branch of the code (starting at the `else` clause) is executed if no `id` argument was present. In this case, we separate the keyword names and values and set the attributes of the instance we are creating. The keyword names and values are then used to construct an SQL statement to insert a new row in the table associated with this `Entity` (highlighted). For example:

```
Car(make="Volvo", model="C30", licenseplate="12-abc-3")
```

Will give us:

```
insert into Car (make,model,licenseplate) values(?,?,?)
```

The question marks are again placeholders for the values we pass to the `execute()` method.

If calling the `execute()` method (highlighted) went well, we initialize the `id` attribute of the instance we are creating with the value of the `lastrowid` attribute. Because we defined the primary key as a `primary key integer autoincrement` column and did not specify it in the insert statement, the primary key will hold a new unique integer and this integer is available as the `lastrowid` attribute.

This is very SQLite-specific and the primary key should be defined in exactly this way for this to hold true. More on this can be found at `http://www.sqlite.org/lang_createtable.html#rowid`

Any `sqlite3.IntegrityError` that might be raised due to the violation of a uniqueness constraint is caught and re-raised as a `ValueError` with slightly more meaningful text.

The `update()` method is used to synchronize an instance with the database. It can be used in two ways: we can alter any attributes of an instance first and then call `update()`, or we may pass keyword arguments to `update()` to let `update()` alter the corresponding attributes and synchronize the instance to the database. These two ways may even be combined. Either way, the database will hold the most current values of all attributes corresponding to a column once the `update()` returns. The following two pieces of code are therefore equivalent:

```
car.update(make='Peugeot')
```

And:

```
car.make='Peugeot'
car.update()
```

Any keyword arguments we pass to `update()` should match a column name, otherwise an exception is raised (highlighted).

Chapter5/entity.py

```
    def update(self,**kw):
        for k in kw:
            if not k in self.__class__.columns :
                raise KeyError("unknown column")
        for k,v in kw.items():
            setattr(self,k,v)
        updates=[]
        values=[]
        for k in self.columns:
            updates.append("%s=?"%k)
            values.append(getattr(self,k))
        updates=",".join(updates)
```

```
            values.append(self.id)
            sql="update %s set %s where %s_id = ?"%(
            self.__class__.__name__, updates, self.__class__.__name__)
            with self.threadlocal.connection as conn:
                  cursor=conn.cursor()
                  cursor.execute(sql, values)
                  if cursor.rowcount != 1 :
                        raise ValueError(
                        "number of updated entities not 1 (%d)" %
                        cursor.rowcount)
```

The column names and the values of the corresponding attributes are then used to construct an SQL statement to update records with these values, but only for the single record whose primary key matches the ID of the instance we are updating. The SQL statement might look like this:

```
update Car set make=?, model=?, licenseplate=? where Car_id = ?
```

The question marks again are placeholders for the values we pass to the `execute()` method.

After we execute this statement, we do a sanity check by validating that the number of affected records is indeed one. Just as for an insert statement, this number is available as the `rowcount` attribute of the cursor object after an update statement (highlighted).

Deleting an instance is implemented by the `delete()` method of the `Entity` class and consists primarily of composing an SQL statement that will delete the record with a primary key equal to the `id` attribute of the instance. The resulting SQL looks like this:

```
delete from Car where Car_id = ?
```

Just like in the `update()` method, we end with a sanity check to verify that just a single record was affected (highlighted). Note that `delete()` will only remove the record in the database, not the Python instance it is called on. If nothing references this object instance, it will be automatically removed by the Python's garbage collector:

Chapter5/entity.py

```
      def delete(self):
            sql="delete from %s where %s_id = ?"%(
            self.__class__.__name__,self.__class__.__name__)
            with self.threadlocal.connection as conn:
                  cursor=conn.cursor()
                  cursor.execute(sql,(self.id,))
                  if cursor.rowcount != 1 :
                        raise ValueError(
                        "number of deleted entities not 1 (%d)" %
                        cursor.rowcount)
```

The final method we encounter is the class method `list()`. This method may be used to retrieve the IDs of all instances of an entity when called without arguments or to retrieve the IDs of instances that match certain criteria passed as arguments. For example:

```
Car.list()
```

Will return a list of IDs of all cars in the database, whereas:

```
Car.list(make='Volvo')
```

Will return the IDs of all the Volvos in the database.

Chapter5/entity.py

```
    @classmethod
    def list(cls,**kw):
            sql="select %s_id from %s"%(cls.__name__,cls.__name__)
            cursor=cls.threadlocal.connection.cursor()
            if len(kw):
                    cols=[]
                    values=[]
                    for k,v in kw.items():
                            cols.append(k)
                            values.append(v)
                    whereclause = " where "+",".join(c+"=?" for c in
cols)
                    sql += whereclause
                    cursor.execute(sql,values)
            else:
                    cursor.execute(sql)
            for row in cursor.fetchall():
                    yield row[0]
```

The implementation is straightforward and starts off with creating an SQL statement to select all IDs from the table (highlighted). An example would be:

```
select Car_id from Car
```

If there were any keyword arguments passed to the `list()` method, these are then used to construct a `where` clause that will restrict the IDs returned to those of the records that match. This `where` clause is appended to our general select statement (highlighted). For example:

```
select Car_id from Car where make=?
```

After invocation of the `execute()` method, we yield all the IDs. By using the `yield` statement, we have identified the `list()` method as a **generator** that will return the IDs found one-by-one rather than in one go. We still can manipulate this generator just like a list if we wish, but for very large result sets, a generator might be a better option as it does consume less memory, for example.

The Relation class

The `Relation` class is used to manage relations between individual instances of entities. If we have `Car` entities as well as `Owner` entities, we might like to define a `CarOwner` class that provides us with the functionality to identify the ownership of a certain car by a specific owner.

Like entities, generic relations share a lot of common functionality: we must be able to create a new relation between two entities, delete a relation, and list related entities given a primary entity, for example, list all owners of a given car or all cars of a certain owner.

Relations are stored in the database in a table, often called a **bridging table**, consisting of records with columns that store the IDs of both related entities. When an application starts using a (subclass of) the `Relation` class, we must verify that the corresponding table exists, and if not, create it.

Time for action – using the Relation class

Let's have a look at how we would like to use our `Relation` class:

Chapter5/carexample2.py

```
from entity import Entity
from relation import Relation

class Car(Entity): pass
class Owner(Entity): pass

Car.threadinit('c:/tmp/cardatabase2.db')
Car.inittable(make="",model="",licenseplate="unique")

Owner.threadinit('c:/tmp/cardatabase2.db')
Owner.inittable(name="")

class CarOwner(Relation): pass

CarOwner.threadinit('c:/tmp/cardatabase2.db')
CarOwner.inittable(Car,Owner)

mycar = Car(make="Volvo",model="C30",licenseplate="12-abc-3")
mycar2 = Car(make="Renault",model="Coupe",licenseplate="45-de-67")
me = Owner(name="Michel")

CarOwner.add(mycar,me)
```

```
CarOwner.add(mycar2,me)

owners = CarOwner.list(mycar)
for r in owners:
   print(Car(id=r.a_id).make,'owned by',Owner(id=r.b_id).name)

owners = CarOwner.list(me)
for r in owners:
   print(Owner(id=r.b_id).name,'owns a',Car(id=r.a_id).make)
```

- Like before, we first define a `Car` class and then an `Owner` class because the `CarOwner` class we define and initialize in the first highlighted lines are only meaningful if the entities in the relation exist. The highlighted lines show that defining and initializing a relation follows the same general pattern as initializing the entities.
- We then create two `Car` entities and an `Owner` and establish a relation between these (second set of highlighted lines).
- The final lines show how we can find and print the owners of a car or the cars belonging to an owner.

Many of these requirements for the `Relation` class are similar to those of the `Entity` class, so when we take a look at the code, some pieces will look familiar.

What just happened?

The first method we encounter is the `threadinit()` class method (the full code is available as `relation.py`). It is identical to the one we encountered in the `Entity` class and should be called once for every thread.

Chapter5/relation.py

```
    @classmethod
    def threadinit(cls,db):
           if not hasattr(cls.threadlocal,'connection') or
cls.threadlocal.connection is None:
                cls.threadlocal.connection=sqlite.connect(db)
                cls.threadlocal.connection.row_factory = sqlite.Row
                cls.threadlocal.connection.execute(
                                                        "pragma
foreign_keys=1")
```

The `inittable()` class method is the method that should be called once when we start an application:

Chapter5/relation.py

```
    @classmethod
    def inittable(cls, entity_a, entity_b,
                              reltype="N:N", cascade=None):
        sql='''create table if not exists %(table)s (
                %(a)s_id references %(a)s on delete cascade,
                %(b)s_id references %(b)s on delete cascade,
                unique(%(a)s_id,%(b)s_id)
        );
        '''%{'table':cls.__name__,
                     'a':entity_a.__name__,'b':entity_b.__name__}
        with cls.threadlocal.connection as conn:
                cursor=conn.cursor()
                cursor.execute(sql)
        cls.columns=[entity_a.__name__,entity_b.__name__]
```

It takes the two classes involved in the relations as arguments to construct a proper SQL statement to create a bridging table if it does not exist yet (highlighted).

For example, `CarOwner.inittable(Car,Owner)` will result in a statement like this:

```
create table if not exists CarOwner (
                Car_id references Car on delete cascade,
                Owner_id references Owner on delete cascade,
                unique(Car_id,Owner_id)
```

There are a couple of interesting things to note here. There are two columns each referring to a table by way of the `references` clause. Because we do not explicitly state *which* column we reference inside the table, the reference is made to the primary key. This is a convenient way to write this down and works because we always define a proper primary key for any table that represents an entity.

Another thing to note is the `on delete cascade` clause. This helps us to maintain something called **referential integrity**. It ensures that when the record that is referenced is deleted, the records in the bridging table that refer to it are deleted as well. This way, there will never be entries in a table that represent a relation that points to non-existing entities. To ensure that this referential integrity checking is actually performed, it is necessary to execute a `pragma foreign_keys = 1` instruction for each connection to the database. This is taken care of in the `threadinit()` method.

Finally, there is a `unique` constraint over both the columns. This effectively ensures that we only maintain, at most, a single entry in this table for each relation between two entities. That is, if I own a car, I can enter this specific relation only once.

If the execution of this statement went well, `inittable()` finishes with storing the names of the entity classes that this relation refers to in the `columns` class variable.

Pop quiz – how to check a class

How can we make sure that the classes we are passing as arguments to the `initdb()` method are subclasses of `Entity`?

Relation instances

The `__init__()` method constructs an instance of a `Relation`, that is, we use it to record the relation between two specific entities.

Chapter5/relation.py

```
def __init__(self,a_id,b_id,stub=False):
        self.a_id=a_id
        self.b_id=b_id
        if stub : return
        cols=self.columns[0]+"_id,"+self.columns[1]+"_id"
        sql='insert or replace into %s (%s) values(?,?)'%(
                self.__class__.__name__,cols)
        with self.threadlocal.connection as conn:
                cursor=conn.cursor()
                cursor.execute(sql,(a_id,b_id))
                if cursor.rowcount!=1:
                        raise ValueError()
```

It takes the IDs of both `Entity` instances that are involved in this specific relation and a `stub` parameter.

The `__init__()` method is not meant to be called directly as it doesn't know nor check whether the IDs passed to it make any sense. It simply stores those IDs if the `stub` parameter is true or inserts a record in the table if it isn't.

Normally, we would use the `add()` method to create a new relationship with all necessary type checking. Separating this makes sense as all this checking is expensive and is unnecessary if we know that the IDs we pass are correct. The `list()` method of the `Relation` class for example retrieves only pairs of valid IDs so that we can use the `__init__()` method without the need for costly additional checks.

The SQL statement that is constructed may look like this for a new `CarOwner` relation:

```
insert or replace into CarOwner (Car_id,Owner_id) values(?,?)
```

If we would try to insert a second relation between the same entities, the `unique` constraint on both columns together would be violated. If so, the `insert or replace` clause would make sure that the insert statement wouldn't fail, but there still would be just one record with these two IDs.

Note that the insert statement could fail for another reason. If either of the IDs we try to insert does not refer to an existing record in the table it refers to, it would fail with an exception `sqlite3.IntegrityError: foreign key constraint failed`.

The final sanity check in the last line is to use the `rowcount` attribute to verify that just one record was inserted.

The `add()` method *does* make sure that the instances passed to it are in the correct order by checking the names of the classes against the names of the columns stored by the `inittable()` method. It raises a `ValueError()` if this is not correct, otherwise it instantiates a new relation by calling the class constructor with both IDs.

Chapter5/relation.py

```
@classmethod
def add(cls,instance_a,instance_b):
       if instance_a.__class__.__name__ != cls.columns[0] :
              raise ValueError("instance a, wrong class")
       if instance_b.__class__.__name__ != cls.columns[1] :
              raise ValueError("instance b, wrong class")
       return cls(instance_a.id,instance_b.id)
```

The `list()` method is meant to return a list of zero or more `Relation` objects.

Chapter5/relation.py

```
@classmethod
def list(cls,instance):
       sql='select %s_id,%s_id from %s where %s_id = ?'%(
              cls.columns[0],cls.columns[1],
              cls.__name__,instance.__class__.__name__)
       with cls.threadlocal.connection as conn:
              cursor=conn.cursor()
              cursor.execute(sql,(instance.id,))
              return [cls(r[0],r[1],stub=True)
                           for r in cursor.fetchall()]
```

It needs to work for both sides of the relation: if we pass a `Car` instance, for example, to the `list()` method of the `CarOwner` class, we should find all records where the `Car_id` column matches the `id` attribute of the `Car` instance.

Likewise, if we pass an `Owner` instance, we should find all records where the `Owner_id` column matches the `id` attribute of the `Owner` instance. But precisely because we gave the columns in the table that represents the relation meaningful names derived from the names of the classes and hence the tables, this is rather straightforward. For example, the SQL constructed for `CarOwner.list(car)` might look like the following:

```
select Car_id,Owner_id from CarOwner where Car_id = ?
```

Whereas the SQL for `CarOwner.list(owner)` would look like the following:

```
select Car_id,Owner_id from CarOwner where Owner_id = ?
```

This is accomplished by referring to the class name of the instance passed as argument (highlighted).

After executing this SQL statement, the results are fetched with the `fetchall()` method and returned as a list of relation instances. Note that this list may be of zero length if there weren't any matching relations.

The last method of note defined for the `Relation` class is the `delete()` method.

Chapter5/relation.py

```
    def delete(self):
        sql='delete from %s where %s_id = ? and %s_id = ?'%(
          self.__class__.__name__,self.columns[0],self.columns[1])
        with self.threadlocal.connection as conn:
            cursor=conn.cursor()
            cursor.execute(sql,(self.a_id,self.b_id))
            if cursor.rowcount!=1:
                raise ValueError()
```

It constructs an SQL delete statement which, in our `CarOwner` example, may look like this:

```
delete from CarOwner where Car_id = ? and Owner_id = ?
```

The sanity check we perform in the last line means that an exception is raised if the number of deleted records is not exactly one.

If there was not exactly one record deleted, what would that signify?

If it would be more than one, that would indicate a serious problem because all the constraints are there to prevent that there is never more than one record describing the same relationship, but if it would be zero, this would probably mean we try to delete the same relationship more than once.

You might wonder why there isn't any method to update a `Relation` object in any way. The reason is that this hardly makes any sense: either there is a relation between two entity instances or there isn't. If we would like to transfer ownership of a car, for example, it is just as simple to delete the relation between the car and the current owner and then add a new relation between the car and the new owner.

Now that we have a simple Entity and Relation framework, let's look at how we can use this to implement the foundation of our books application.

Time for action – defining the Books database

The next step is to create a module `booksdb.py` that uses the `entity` and `relation` modules to construct a data model that can be used conveniently by the delivery layer (the parts of the web application that deal with providing content to the client). We therefore have to define `Book`, `Author`, and `User` entities as well as a `BookAuthor` relation and a `UserBook` relation.

We will also provide some functions that are bit more high-level, for example, a `newbook()` function that checks whether a book with a given title already exists and that only creates a new `Book` instance if the authors are different (presumably because they wrote a book with the same title).

Having a separate layer that models data in terms that are meaningful in the context makes it easier to understand what is going on. It also keeps the delivery layer less cluttered and therefore easier to maintain.

What just happened?

After importing the `Entity` and `Relation` class, the first thing we do is define the appropriate entities and relations (the full code is available as `booksdb.py`). The first function we encounter is `threadinit()` (highlighted). It is a convenience function that calls all the individual `threadinit()` methods of the different entities and relations we have defined:

Chapter5/booksdb.py

```
from entity import Entity
from relation import Relation

class Book(Entity):
```

```
    pass
class Author(Entity):
    pass
class User(Entity):
    pass
class BookAuthor(Relation):
    pass
class UserBook(Relation):
    pass
def threadinit(db):
    Book.threadinit(db)
    Author.threadinit(db)
    User.threadinit(db)
    BookAuthor.threadinit(db)
    UserBook.threadinit(db)
```

Likewise, the `inittable()` function is a convenience function that calls all the necessary `inittable()` methods:

Chapter5/booksdb.py

```
def inittable():
    Book.inittable(title="",isbn="unique",publisher="")
    Author.inittable(name="")
    User.inittable(userid="unique not null")
    BookAuthor.inittable(Book,Author)
    UserBook.inittable(User,Book)
```

It defines a `Book` as a subclass of `Entity` having a `title`, a `unique isbn`, and a `publisher` attribute. An `Author` is defined as a subclass of `Entity` with just a `name` attribute and a `User` as an `Entity` with just a `userid` that must be unique and cannot be null. Also, the relations that exist between `Book` and `Author`, and `User` and `Book` are initialized here.

The `newbook()` function should be used to add a new book to the database:

Chapter5/booksdb.py

```
def newbook(title,authors,**kw):
    if not isinstance(title,str) :
           raise TypeError("title is not a str")
    if len(title)<1 :
           raise ValueError("title is empty")
    for a in authors :
           if not isinstance(a,Author) :
```

```
                raise TypeError("authors should be of type Author")
    bl=list(Book.list(title=title,**kw))
    if len(bl) == 0:
                b=Book(title=title,**kw)
    elif len(bl) == 1:
                b=Book(id=bl[0])
    else:
          raise ValueError("multiple books match criteria")
    lba=list(BookAuthor.list(b))
    if len(authors):
          lba=[Author(id=r.b_id) for r in lba]
          for a in authors:
                known=False
                for a1 in lba:
                      if a.id == a1.id :
                            known=True
                            break
                if not known:
                      r=BookAuthor.add(b,a)
    return b
```

It takes a `title` argument and a list of `Author` objects and any number of optional keywords to select a unique book if the title is not sufficient to identify a book. If a book with the given title and additional keywords cannot be found, a new `Book` object is created (highlighted). If more than one book is found that matches the criteria, an exception is raised.

The next step is to retrieve a list of authors associated with this book. This list is used to check if any author in the list of authors passed to the `newbook()` function is not already associated with this book. If not, this new author is added. This ensures we do not attempt to associate an author more than once with the same book, but it also makes it possible to add authors to the list of authors associated with an existing book.

The `newauthor()` function verifies that the name passed as an argument is not empty and is indeed a string (highlighted):

Chapter5/booksdb.py

```
def newauthor(name):
    if not isinstance(name,str) :
          raise TypeError("name is not a str")
    if len(name)<1 :
          raise ValueError("name is empty")
    al=list(Author.list(name=name))
    if len(al) == 0:
```

```
                a=Author(name=name)
    elif len(al) == 1:
                a=Author(id=al[0])
    else:
            raise ValueError("multiple authors match criteria")
    return a
```

Then it checks whether an author with such a name already exists. If it doesn't, a new `Author` object is created and returned. If only one `Author` was found, that one is returned without creating a new one. If the same name matched more than one `Author`, an exception is raised because our current data model does not provide the notion of more than one author with the same name.

An application to register books is most often used to see if we already own a book. A function to list books matching a set of criteria should therefore be quite flexible to provide the end user with enough functionality to make finding books as simple as possible, even if the books number is in the thousands.

The `listbooks()` function tries to encapsulate the necessary functionality. It takes a number of keyword arguments used to match any number of books. If the `user` argument is present, the results returned are limited to those books that are owned by that user. Likewise, the `author` argument limits the results to books by that author. The `pattern` argument may be a string that limits the books returned to those whose title contains the text in the `pattern` argument.

Because the number of books matching the criteria could be very large, `listbooks()` takes two additional parameters to return a smaller subset. This way, the delivery layer can offer the list of results in a page-by-page manner. The `offset` argument determines the start of the subset and `limit` of the number of results returned. If `limit` is `-1`, all results starting at the given `offset` are returned. For example:

```
booksdb.listbooks(user=me,pattern="blind",limit=3)
```

Would return the first three books I own that have the text blind in their title.

Given these requirements, the implementation of `listbooks()` is rather straightforward:

Chapter5/booksdb.py

```
def listbooks(user=None,author=None,offset=0,limit=-1,pattern=""):
    lba={}
    lbu={}
    if not user is None:
            if not isinstance(user,User):
                    raise TypeError("user argument not a User")
            lbu={r.b_id for r in UserBook.list(user)}
```

```
    if not author is None:
        if not isinstance(author,Author):
            raise TypeError("author argument not an Author")
        lba={r.a_id for r in BookAuthor.list(author)}

    if user is None and author is None:
        lb={b for b in Book.list()}
    else:
        if len(lbu)==0 : lb=lba
        elif len(lba)==0 : lb=lbu
        else : lb = lba & lbu

    books = [Book(id=id) for id in lb]
    books = sorted(books,key=lambda book:book.title.lower())
    if pattern != "" :
        pattern = pattern.lower()
        books = [b for b in books
                            if b.title.lower().find(pattern)>=0 ]
    if limit<0 :
        limit=len(books)
    else:
        limit=offset+limit
    return len(books),books[offset:limit]
```

It starts by checking that any `user` argument is indeed an instance of a `User` entity and then finds all books owned by this user (highlighted) and converts this list to a **set**. It checks any `author` argument in a similar way. If neither an author nor a user was specified, we simply retrieve a list of all books (highlighted) and convert it to a set as well.

Working with sets is convenient, as sets will never contain duplicates and can easily be manipulated. For example, if we have a non-empty set of books associated with an author and a non-empty set of books owned by a user, we can obtain the **intersection** (that is, books both owned by the given owner and written by the given author) with the `&` operator.

Either way, we end up with a list of book IDs in `lb`. This list of IDs is then converted to `Book` objects and sorted on the title to ensure consistent results when dealing with offsets (highlighted). The next step is to reduce the number of results to those books whose title contains the text in the `pattern` argument.

All this matching, sorting, and filtering could have been done with SQL as well and probably in a more efficient manner too. However, this would mean the SQL would be rather complicated and we would ruin the clear distinction between the low-level database operations defined in the `entity` and `relation` modules and the more high-level operations defined here in `booksdb`. If efficiency was more important, that would be a valid argument, but here we choose for a clear separation to aid understanding, as Python is a lot easier to read than SQL.

All that is left now is to return the proper slice from the list of books based on the `offset` and `limit` arguments, as shown in the last line of the `listbooks()` function. Note that we actually return a tuple, the first item being the total number of matching books, the second item the actual list of matching books. This makes it simple to present information to the end user like 'showing items 20-29 of 311'.

The `listauthors()` function either returns a list of authors associated with a book if a `book` argument is given or a list of all authors:

Chapter5/booksdb.py

```
def listauthors(book=None):
    if not book is None:
           if not isinstance(book,Book):
                  raise TypeError("book argument not a Book")
           la=[r.b_id for r in BookAuthor.list(book)]
    else:
           la=Author.list()
    return [Author(id=id) for id in la]
```

It does make sure that any `book` argument is indeed an instance of a `Book` entity.

`checkuser()` may be called to see if there already exists a user with the given username and if not creates one:

Chapter5/booksdb.py

```
def checkuser(username):
    users=list(User.list(userid=username))
    if len(users):
           return User(id=users[0])
    return User(userid=username)
```

Any user that uses this application should have a corresponding `User` entity, if he/she wants to be able to register his/her ownership of a book. This function makes sure this is possible.

Note that our application does not *authenticate* a user at this level, that is left to the delivery layer as we will see. Any authentication database the delivery layer uses is completely separate from the `User` entity in our books database. The delivery layer may, for example, use the system password database to authenticate a user and pass the username to this layer if the authentication was successful. If, at that point, the user does not yet exist in our books database, we can make sure he does by calling the `checkuser()` function.

The `addowner()` and `delowner()` functions are used to establish or remove a specific ownership relation between a book and a user. Both are thin wrappers around the underlying methods in the `Relation` class, but add some additional type checking.

Chapter5/booksdb.py

```
def addowner(book,user):
    if not isinstance(book,Book):
           raise TypeError("book argument not a Book")
    if not isinstance(user,User):
           raise TypeError("user argument not a User")
    return UserBook.add(user,book)

def delowner(book,user):
    if not isinstance(book,Book):
           raise TypeError("book argument not a Book")
    if not isinstance(user,User):
           raise TypeError("user argument not a User")
    UserBook(user.id,book.id,stub=True).delete()
```

This foundation will be put to good use in the next section where we will implement the delivery layer.

Pop quiz – how to select a limited number of books

How would you select the third page of 10 books from a list of all books in the database?

Have a go hero – cleaning up the books database

`booksdb.py` lacks a `delbooks()` function because we won't be providing this functionality in our final application. It is not a disadvantage to just remove ownership and leave the book as is, even if it doesn't have any owners because other users may register ownership by referring to this existing book without the need to enter it again. However, occasionally we might want to clean up the database. How would you implement a function that removes all books without an owner?

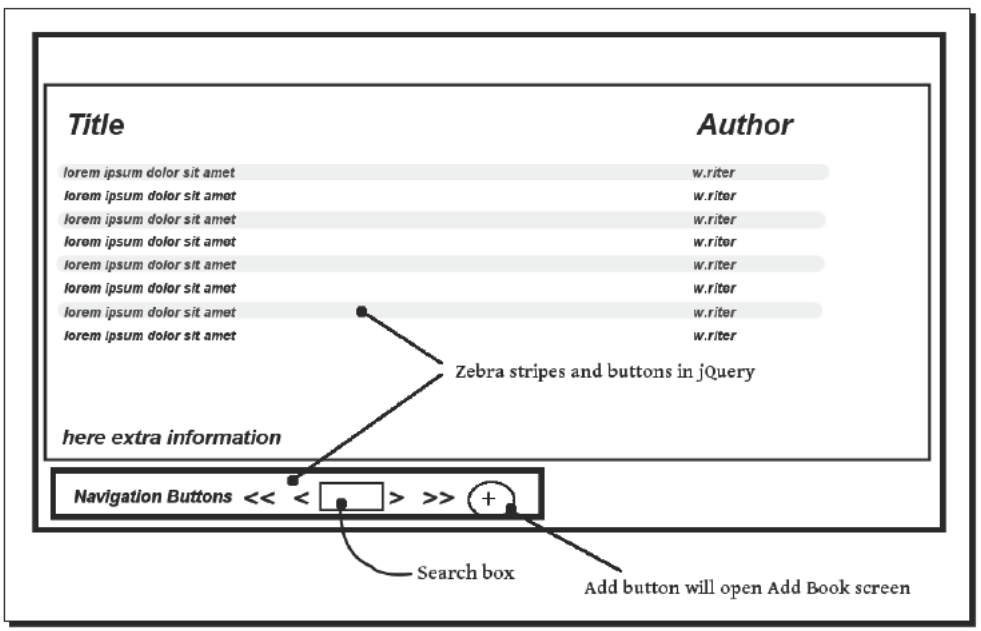
The delivery layer

Because we laid a substantial foundation with the `entity`, `relation`, and `booksdb` modules, we can now cleanly separate the delivery layer from the rest of the application. The delivery layer consists of just a couple of CherryPy applications. To authenticate a user, we will reuse the logon application we encountered in previous chapters and the rest of the application consists of a single `Books` class with the necessary methods to provide two main screens: a navigable and filterable list of books and a screen to add new books to the database.

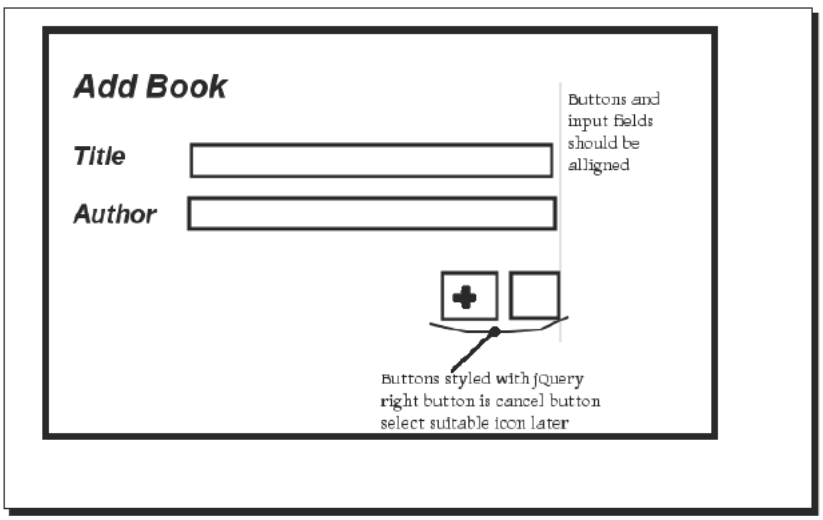
Time for action – designing the delivery layer

To design these screens it is often convenient to make some drawings to have a visual representation of the screen. This makes it a lot easier to discuss functionality with your client.

There exist a number of good applications to assist you with drawing up some mock ups (for example, Microsoft's Expression Blend or Kaxaml `http://kaxaml.com/`) but often, especially in the early stages of designing an application, a simple drawing will do, even if it's hand drawn. The illustrations show the sketches I used in making a rough draft, both done with the GIMP (`http://www.gimp.org/`):



The first image is a sketch of the screen that lets the user interact with a list of books, the second image shows what a screen to add a new book could look like.



Such images are easy to print and annotate during a discussion without the need for a computer application, all you need is a pen or pencil. Another useful designing technique is to draw some outline on a whiteboard and add details while you discuss functionality. At the end of the session, you can take a picture of the whiteboard with your cell phone and use that as a starting point. The design will likely change anyway during the development and starting with this simple approach saves a lot of effort that might have to be undone later on.

When we take a look at the design for the screen that lists the books we see immediately that the key functionality is all in the **button bar**. Notably, we will have to implement functionality to:

- Show a list of books
- Page forward and backward through this list
- Limit the list of books to those owned by the current user
- Filter the list of books on words occurring in the title

The screen to add a new book is deceptively simple. The user must be able to enter a title and an author to indicate he owns a book, but this means that in the background, we have at least the following scenarios to check:

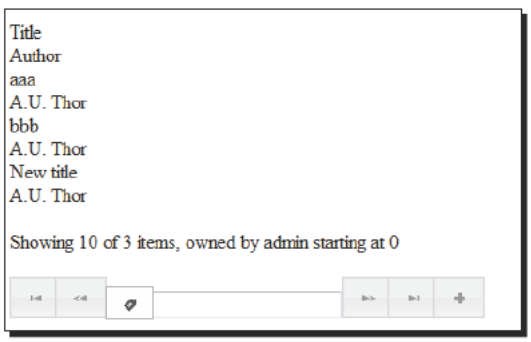
- There is no book in the database with the given title
- There is a book with the given title but without the given author
- The combination of book and author is known

In the first situation, we have to create a new `Book` entity, and possibly a new `Author` entity if the author is unknown.

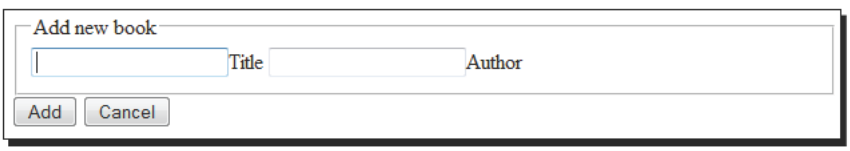
In the second situation, we will create a new `Book` entity as well, because it is very well possible that different authors write books with the same title. In a more elaborate application, we might be able to make a distinction based on the ISBN.

In the last situation, we do not have to create a new `Book` or an `Author` entity, but we still have to make sure that we register ownership of that specific book.

The final requirement is one of convenience to the user. If there are many users entering books in the database, chances will grow that if someone registers a new book he/she owns, that book is already present in the database. To save the user some typing, it would be nice if we could present the user with a list of possible titles and authors while he/she types along. This is called auto completion and is fairly straightforward to implement with a combination of jQuery UI and Python.



When `booksweb.py` is started, the list of books will look like the preceding image and the page to add a new book is shown next. We will enhance these looks in the last section of this chapter, but we focus on the implementation of the delivery layer in `booksweb.py` first.



Auto completion is a close companion of client-side input validation. By presenting the user with a list of possibilities to choose from, we lower the risk of a user entering a similar title with a slightly different spelling. Of course there are some additional checks to make: a title may not be empty, for example. If the user does make an erroneous entry, there should also be some sort of feedback so he/she can correct his mistake.

Of course, client-side validation is a useful tool to enhance the user experience, but it doesn't protect us from malicious attempts to corrupt our data. Therefore, we have implemented some server-side checks as well.

What just happened?

We start off by creating a global variable that holds the basic HTML that we will use both in the booklist screen as well as in the add book screen (the full code is available as `booksweb.py`):

Chapter5/booksweb.py

```
with open('basepage.html') as f:
    basepage=f.read(-1)
```

We read it in from a separate file instead of storing it inside the module as a string. Storing the HTML in a separate file makes it a lot easier to edit because the editor can use syntax highlighting for HTML instead of just marking it as a string literal in Python. The file is available as `basepage.html`:

Chapter5/basepage.html

```
<html><head><title>Books</title>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.
min.js" type="text/javascript"></script>
<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.3/
jquery-ui.min.js" type="text/javascript"></script>
<link rel="stylesheet" href="http://ajax.googleapis.com/ajax/libs/
jqueryui/1.8.3/themes/smoothness/jquery-ui.css" type="text/css"
media="all" />
<link rel="stylesheet" href="/books.css" type="text/css" media="all"
/>
</head><body>
<div id="content">%s</div>
<script src="/booksweb.js" type="text/javascript"></script>
</body>
</html>
```

This time, we choose to incorporate all external libraries from Google's content delivery network (highlighted).

You might not want to depend on an external agency for your production application, but for development, this is an advantage as you don't have to lug around those files. But even in a production environment, this choice may make sense as this option will reduce the number of requests made to your server and minimize the bandwidth. Likewise, we refer to the cascading style sheets and accompanying files for our chosen theme (Smoothness) on Google's content delivery network.

Besides Google, a number of other large players offer a Content Delivery Network (CDN) that you may use. Even Microsoft (`http://www.asp.net/ajaxlibrary/cdn.ashx`) offers free access to the jQuery and jQuery UI libraries on its CDN.

The head section also contains a link to an additional style sheet `books.css` that will be used to tweak the layout and style of the elements that are not jQuery UI widgets.

The body is a single `<div>` element with a `%s` placeholder to be filled with different relevant markup for the booklist and new book pages, followed by a `<script>` tag that will provide other script elements within the specific pages with common functionality.

`booksweb.py` continues with the definition of the `Books` class that will act as the central application in the CherryPy framework for this application.

Chapter5/booksweb.py

```
class Books():
    def __init__(self,logon,logoffpath):
            self.logon=logon
            self.logoffpath=logoffpath

    @cherrypy.expose
    def index(self):
            username = self.logon.checkauth()
            return basepage % '<div id="booklist"></div>'
```

The `index()` function serves `basepage.html` with a single `<div>` element that will hold the content.

The `list()` method will be called from JavaScript functions defined in `booksweb.js` once it loads and will be used to fill the content `<div>` initially, as well as refresh the contents of this div when the buttons in the navigation button bar are pressed.

Before we examine the `list()` and `addbook()` methods, let's have a look at the JavaScript in `booksweb.js` to see how these methods are invoked from the AJAX calls in the client (the full JavaScript code is available as `booksweb.js`).

Chapter5/booksweb.js

```
$.ajaxSetup({cache:false,type:"GET"});
```

The first activity we encounter in `booksweb.js` is setting global defaults for all AJAX calls. We disable caching to make sure the browser executes an AJAX call every time we ask it to, without checking if it made a call to the same URL earlier, because otherwise we would not actually refresh the content of our list of books.

For debugging purposes, we also make sure every AJAX call uses the HTTP `GET` method because arguments to a `POST` call are not logged normally while arguments to a `GET` are part of the URL of the request.

The `prepnavbar()` function we encounter next is our workhorse: every time we make the URL `/books/list` get a list of books, `prepnavbar()` is called once the request is completed.

Chapter5/booksweb.js

```
function prepnavbar(response, status, XMLHttpRequest){
    $("#firstpage").button({
                text: false,
                icons: {
                       primary: "ui-icon-seek-start"
                }
    });
    $("#previouspage").button({
                text: false,
                icons: {
                       primary: "ui-icon-seek-prev"
                }
    });
    $("#mine").button({
                text: false,
                icons: {
                       primary: "ui-icon-tag"
                }
    });
    $("#nextpage").button({
                text: false,
                icons: {
                       primary: "ui-icon-seek-next"
                }
    });
    $("#lastpage").button({
                text: false,
                icons: {
                       primary: "ui-icon-seek-end"
                }
    });
    $("#addbook").button({
                text: false,
                icons: {
                       primary: "ui-icon-plusthick"
```

```
                }
    });
    t=$("#toolbar").buttonset();
    $("span",t).css({padding:"0px"});
    $(".bookrow:odd").addClass('oddline');
};
$("#booklist").load('/books/list',prepnavbar);$("#booklist").load('/
books/list',prepnavbar);
```

The HTML returned by `/books/list` not only contains the matching books, but also the navigation buttons themselves together with additional information on the number of matching books returned. These navigation buttons are not yet styled and configuring this is the task of the `prepnavbar()` function.

It styles each button (except for the input button that is used to filter on text) as a jQuery UI button widget without text but with an appropriate icon. It also adds the `oddline` class to each odd row of the `bookrow` class, so we can refer to this class in our style sheet to give it distinctive zebra stripes, for example.

When `booksweb.js` is executed, the content of the page consists of an empty `<div>`. This `<div>` element is filled with HTML returned by calling the `/books/list` URL with parameters (last line). The `prepnavbar()` function is passed as the second argument to the `load()` method and will be called once loading the data is completed.

The remaining part of `booksweb.js` is filled with adding live click handlers to all navigation buttons.

Chapter5/booksweb.js

```
function getparams(){
    var m=0;
    // apparently the checked attr of a checkbox is magic:
// it returns true/false, not the contents!
    if ( $("#mine").attr("checked")==true ) { m = 1}
    return { offset:Number($("#firstid").text()),
             limit:Number($("#limitid").text()),
                  filter:$("#filter").val(),
                  mine:m
              };
};
$("#mine").live('click',function(){
    // this function is fired *after* the click
// toggled the checked attr
    var data = getparams();
    if (data.mine) {
```

```
            $("#mine").removeAttr("checked");
        } else {
            $("#mine").attr("checked","yes");
        }
        $("#booklist").load('/books/list',data,prepnavbar);
        return true;
    });
    $("#firstpage").live('click',function(){
        var data = getparams();
        data.offset=0;
        $("#booklist").load('/books/list',data,prepnavbar);
        return true;
    });
    $("#previouspage").live('click',function(){
        var data = getparams();
        data.offset -= data.limit;
        if(data.offset<0){ data.offset=0;}
        $("#booklist").load('/books/list',data,prepnavbar);
        return true;
    });
    $("#nextpage").live('click',function(){
        var data = getparams();
        var n=Number($("#nids").text())
        data.offset += data.limit;
        if(data.offset>=n){ data.offset=n-data.limit;}
        if(data.offset<0){ data.offset=0;}
        $("#booklist").load('/books/list',data,prepnavbar);
        return true;
    });
    $("#lastpage").live('click',function(){
        var data = getparams();
        var n=Number($("#nids").text())
        data.offset = n-data.limit;
        if(data.offset<0){ data.offset=0;}
        $("#booklist").load('/books/list',data,prepnavbar);
        return true;
    });
    $("#filter").live('keyup',function(event){
        if (event.keyCode == '13') {
            event.preventDefault();
            data = getparams();
            data.offset=0;
            $("#booklist").load('/books/list',data,prepnavbar);
        }
```

```
    return true;
});
$("#addbook").live('click',function(){
    window.location.href="/books/addbook";
    return true;
});
```

A live handler will be attached to any element that matches its selector, even elements that are not present yet in the documents. This will ensure that when we reload the list of books complete with new navigation buttons, the click handlers we define here will be bound to these new buttons as well.

Each of these handlers call the `getparams()` function to retrieve the information contained in the `<p>` element with the `id="info"`. This data is returned as a JavaScript object that may be passed to the `load()` method. The `load()` method will append the attributes in this object as parameters to the URL it calls. The information in the object reflects the currently listed books and each handler modifies this data according to its function.

For example, the handler for the `#firstpage` button (highlighted) modifies the `offset` attribute. It simply sets it to zero before calling `/books/load` to retrieve the first set of books.

The handler for the `#previouspage` button subtracts the value of the `limit` attribute from offset to get the previous page full of books, but makes sure that the offset is not smaller than zero. The handlers for the other clickable buttons perform similar actions before calling `/books/load`.

The exception is the handler for the `#mine` button, that does not manipulate offsets but toggles the `checked` attribute.

The `#pattern` input element is different as well. It doesn't act on a click, but reacts on pressing the *return* key. If that key is pressed, it also calls `getparams()` just like the other handlers. The object retrieved this way will also contain a `pattern` attribute, which holds the value of the `#pattern` input element that was just entered by the user. The `offset` attribute is set to zero to ensure that when we pass on a new pattern value, we start viewing the resulting list at the start.

Let's return to the server-side in `booksweb.py` and see how the `list()` method is implemented.

Chapter5/booksweb.py

```
@cherrypy.expose
def list(self,offset=0,limit=10,mine=1,pattern="",_=None):
       username = self.logon.checkauth()
       userid=booksdb.checkuser(username)
       try:
```

```
                offset=int(offset)
                if offset<0 : raise ValueError("offset < 0")
            except ValueError:
                raise TypeError("offset not an integer")
            try:
                limit=int(limit)
                if limit<-1 : raise ValueError("limit < -1")
            except ValueError:
                raise TypeError("limit not an integer")
            try:
                mine=int(mine)
            except ValueError:
                raise TypeError("mine not an integer")
            if not mine in (0,1) :
                raise ValueError("mine not in (0,1)")
            if len(pattern)>100 :
                raise ValueError("length of pattern > 100")
            # show titles
            yield '<div class="columnheaders"><div class="title">Title</
div><div class="author">Author</div></div>'
            # get matching books
            if mine==0 : userid=None
            n,books = booksdb.listbooks(user=userid,
                offset=offset,limit=limit,pattern=pattern)
            # yield them as a list of divs
            for b in books:
                a1=booksdb.listauthors(b)[0]
                yield '''<div id="%d" class="bookrow">
<div class="title">%s</div>
<div class="author">%s</div>
</div>'''%(b.id,b.title,a1.name)
            # yield a line of navigation buttons
            yield '''<div id="navigation">
<p id="info">Showing
<span id="limitid">%d</span> of
<span id="nids">%d</span> items,
 owned by <span id="owner">%s</span> starting at
<span id="firstid">%d</span>
</p>
<div id="toolbar">
<button id="firstpage" value="First">First</button>
<button id="previouspage" value="Previous">Prev</button>
<input id="mine" type="checkbox" %s /><label for="mine">Mine</label>
<input id="pattern" type="text" value="%s" />
```

```
<button id="nextpage" value="Next" >Next</button>
<button id="lastpage" value="Last" >Last</button>
<button id="addbook" value="Add">Add</button>
</div>
</div>'''%(limit,n,username if mine else "all",
             offset,'checked="yes"'if mine else "", pattern)
```

The `list()` method takes a number of keyword arguments to determine which books to return. It doesn't return a complete HTML page but just a list of `<div>` elements representing the selection of books together with some additional information on the number of books selected and the button elements used to browse through the list:

- The `offset` argument determines where the list of matching books will start. Counting starts at 0.
- The `limit` argument determines the number of matching books to return. This is a maximum, less books will be returned if they are not available. When we have 14 matching books, an offset of 10, with a limit of 10, will return 10 books through 13.
- If the `mine` argument is non-zero, the list of matching books is limited to the ones owned by the user issuing the request.
- if the `pattern` argument is not an empty string, the list of matching books is limited to the ones that contain the pattern string in their title.
- The _ (underscore) argument is ignored. We configured our AJAX calls not to be cached (in `booksweb.js`) and jQuery prevents caching by appending each time an _ argument with a random value to the URL it requests. The URL will look different each time to the browser this way, and this will prevent caching.

The implementation of the `list()` method starts off by validating that the user is logged in and then retrieving the corresponding `User` object. The next steps systematically validate the arguments passed to the method and raise a `ValueError` or `TypeError` if something fails to validate. The `offset` argument, for example, should be larger or equal to zero (highlighted).

Once the arguments are validated, these are handed off to the `booksdb.listbooks()` function, that will take care of the actual selection and will return a tuple consisting of the number of matching books and the actual list of books sorted on their title.

This list of books is used to step through and generate the appropriate HTML markup. For each book, we fetch the authors of the book (highlighted) and then yield a string with HTML markup. This HTML contains the title of the book and the name of the first author. If there is more information we would like to present, for example, the ISBN of the book, we could easily add it here. By using `yield` to return the results one-by–one, we save ourselves the trouble of constructing a complete string first before returning it in one go.

The final `yield` statement contains a `<div>` element with the `id="navigation"`. We choose to return the complete navigation markup, including buttons, to enable us to easily set the values of these buttons. The pattern `<input>` element, for example, should display the current text we filter on. We could pass this as separate information and use client-side JavaScript to set these values but this would complicate the JavaScript quite a bit.

Still, the `offset` and `limit` values together with the total number of matching books is returned inside a `<p>` element. This serves two goals: we can display this as an informational message to the user, but it is also necessary information for the navigation buttons to function.

Time for action – adding a new book

The screen to add a new book to the database is a simple form. What we need to implement is:

- Some HTML to make it possible to display the form
- A method in our CherryPy application that will produce this HTML
- A method to process the input once this form is submitted

There is no need to implement two different methods here because based on the arguments passed to the method we can decide whether to return a form or to process the submitted contents of the same form. Although it may be considered bad form to design a method to do two things, it does keep related functionality together.

What just happened?

The `addbookform` class variable contains the template that refers to a number of string variables to interpolate. There is also a `<script>` element to add some extra functionality that we examine later:

Chapter5/booksweb.py

```
    addbookform='''<div id="newbook">
<form action="addbook" method="get">
<fieldset><legend>Add new book</legend>
<input name="title" id="title" type="text" value="%(title)s"
%(titleerror)s />
<label for="title">Title</label>
<input name="author" id="author" type="text" value="%(author)s"
%(authorerror)s />
<label for="author">Author</label>
</fieldset>
<div class="buttonbar">
<button name="submit" type="submit" value="Add">Add</button>
```

```
<button name="cancel" type="submit" value="Cancel">Cancel</button>
</div>
</form>
<div id="errorinfo"></div>
</div>'''
```

The `addbook()` method itself is used both to display the initial screen and to process the results, that is, it acts as the target of the `<form>` element's action attribute and processes the values from the various `<input>` and `<button>` elements.

All arguments are therefore keyword arguments with default values. If they are all missing, `addbook()` will construct an empty form, otherwise it will check and process the information. In the latter case, there will be two possible scenarios: the values are ok, in which case a new book will be added and the user will be returned to the page with the book listing, or one or more of the values are not ok, in which case, the form is presented again, with suitable error markings, but with the entered values still in place for the user to edit.

Chapter5/booksweb.py

```
    @cherrypy.expose
    def addbook(self,title=None,author=None,submit=None,cancel=None):
        username = self.logon.checkauth()
        userid=booksdb.checkuser(username)
        if not cancel is None: raise cherrypy.HTTPRedirect("/books")
        data=defaultdict(str)
        if submit is None:
            return basepage%(Books.addbookform%data)
        if title is None or author is None:
            raise cherrypy.HTTPError(400,'missing argument')
        data['title']=title
        data['author']=author
        try:
            a=booksdb.newauthor(author)
            try:
                b=booksdb.newbook(title,[a])
                booksdb.addowner(b,userid)
                raise cherrypy.HTTPRedirect("/books")
            except ValueError as e:
data['titleerror']= 'class="inputerror ui-state-error"
title="%s"'%str(e)
        except ValueError as e:
data['authorerror']= 'class="inputerror ui-state-error"
title="%s"'%str(e)
        return basepage%(Books.addbookform%data)
```

The `addbook()` method first verifies if the user is logged in, and if so, fetches the corresponding `User` object. The next step is to check if the cancel button contained in the form was clicked, in which case the `cancel` argument will contain a value and the user will be redirected to the list of books (highlighted).

Next, we create a default dictionary that will return an empty string for every missing key that is accessed. This default dictionary will be used as interpolation data for strings in `addbookform`. This way we can set a number of interpolation variables if we want to (for example, `%(title)s` in the `value` attribute of the `<input>` element for a title), but if we leave anything out, it will be automatically replaced by an empty string.

If the `submit` argument is equal to None, this means it wasn't present, so `addbook()` was called to display just the empty form and that is just what is done (highlighted). Because `data` contains no keys at all at this moment, *all* interpolation variables will yield an empty string resulting in an empty form.

If the `submit` argument was not None, we are processing the values in the form. First we perform a sanity check. If either the `title` or the `author` argument is missing, we raise an exception (highlighted). Even if the user failed to enter either of them, the corresponding values would be present in the arguments but as empty strings. So, if either of these arguments is missing completely, this cannot be the result of a user action and therefore it is sensible to raise an exception.

If both arguments are present, we save them in the default dictionary so that we can redisplay them as default values in the form, if we need to present the form again.

The next step is to try the `newauthor()` function from the `booksdb` module. It either returns a valid `Author` object (because we already know the author or a new one was created) or it raises an exception. Such an exception is caught and the error text is added to the `authorerror` key in the dictionary together with some HTML class attributes that will enable us to display the corresponding `<input>` element in a suitable manner to indicate the error condition.

Once we have a valid `Author` object, the same approach is used to retrieve a valid `Book` object. This may fail (mainly if the `title` argument is an empty string) in which case we set the `titleerror` key in the dictionary.

We end with establishing an ownership relation between the user and the book with a call to the `addowner()` function and then redirect the user to the page that lists the books.

If anything goes wrong, we catch some exception and we end up at the return statement which will return the form again, only this time the dictionary will hold some keys that will be interpolated, resulting in suitable default values (for example, if a `title` argument was empty, but the `author` argument wasn't, the user doesn't have to enter the name of the author again) and information on the errors encountered.

All this string interpolation business might be a bit daunting, so let's have a brief look at an example. The definition for the title `<input>` element in the `addbookform` variable looks like this:

```
<input name="title" id="title" type="text" value="%(title)s"
%(titleerror)s />
```

If we want to present the user with an empty form, the string is interpolated with a default dictionary that holds no keys. The references `%(title)s` and `%(titlerror)s` will therefore come up with empty strings resulting in:

```
<input name="title" id="title" type="text" value=""  />
```

A plain `<input>` element without a default value.

Now if something went wrong with locating or creating an author, the dictionary would hold a `title` key but no `titleerror` key (but it would have an `authorrerror` key). Assuming that the title entered by the user was "A book title", the resulting interpolation would therefore look like this:

```
<input name="title" id="title" type="text" value="A book title"  />
```

Finally, if there was an error with the title, for example, because no book title was entered by the user, both the `title` key would be present (albeit, in this case, as an empty string) and the `titleerror` key. The value of the `titleerror` key holds both the error message as an HTML `title` attribute together with an HTML `class` attribute that looks like this:

```
class="inputerror ui-state-error" title="title is empty"
```

So the final interpolation would result in:

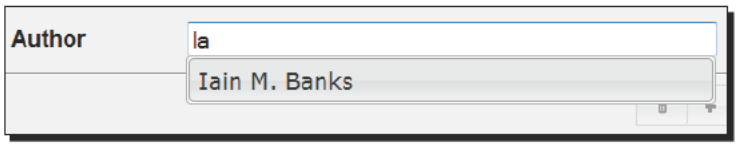
```
<input name="title" id="title" type="text" value="" class="inputerror
ui-state-error" title="title is empty" />
```

Auto completion

When we presented the HTML markup for the page that shows a form to add a new book, we skipped over the `<script>` element at the end. That script element is used to augment the title and author `<input>` elements with **auto completion**.

Time for action – using input fields with auto completion

With the `<script>` element in place, the input elements can now retrieve possible completions with an AJAX call. Now, when we enter a few characters in an input field, we are presented with a list of choices, as shown in the image:



Let's look in some detail at how this is implemented in, remarkably, a few lines of JavaScript.

What just happened?

If we look at the code again, we see that we call the `autocomplete()` method on both the `#title` and `#author` `<input>` elements, but each with a different source argument. The **autocomplete widget** in jQuery UI is very versatile and simple to apply (The code shown is part of `booksweb.py`, but we skipped this earlier):

Chapter5/booksweb.py

```
<script>
$("#title" ).autocomplete({ source:'/books/gettitles',
                                                minLength:2}).focus();
$("#author").autocomplete({ source:'/books/getauthors',
                                                minLength:2});
</script>
```

The `source` attribute of the options object we pass to the `autocomplete()` method holds a URL that will be used to retrieve a list of possible completions for the characters entered so far.

The `minLength` attribute ensures that we only start looking for possible completions once the user has entered at least two characters, otherwise the list would probably be enormous and not much of a help. Note that it is still possible to enter a completely new value into an input field. The user is not obliged to pick an item from the list presented and can keep on typing.

The autocomplete widget adds the text entered so far as the `term` argument to the `source` URL. When the user has typed 'foo' in the `#author` `<input>` element, this will result in a call to a URL like `/books/getauthors?term=foo&_=12345678`.

This means that the `gettitles()` and `getauthors()` methods will both take a `term` argument (and an _ (underscore) argument to ensure nothing is cached):

Chapter5/booksweb.py

```
@cherrypy.expose
    def getauthors(self,term,_=None):
           return json.dumps(booksdb.getauthors(term))

    @cherrypy.expose
    def gettitles(self,term,_=None):
           titles=json.dumps(booksdb.gettitles(term))
           return titles
```

Both methods simply pass on the request to the corresponding `booksdb` functions, but because autocomplete widgets expect the result as a JSON encoded string, we convert the list with the `json.dumps()` function before returning it:

Chapter5/booksdb.py

```
def gettitles(term):
    titles=Book.getcolumnvalues('title')
    re=compile(term,IGNORECASE)
    return list(takewhile(lambda x:re.match(x),
                          dropwhile(lambda x:not re.match(x),titles)))

def getauthors(term):
    names=Author.getcolumnvalues('name')
    re=compile(term,IGNORECASE)
    return list(takewhile(lambda x:re.match(x),
                          dropwhile(lambda x:not re.match(x),names)))
```

The functions `getauthors()` and `gettitles()` in `booksdb.py` could have simply retrieved a list of `Author` or `Book` objects respectively and extracted the `name` or `title` attributes. This would have been fairly slow, however, as creating potentially a lot of objects is costly in terms of processing power. Moreover, since we are really interested in just a list of strings and not in whole objects, it is worthwhile to implement a `getcolumnvalues()` method in the `Entity` class:

Chapter5/entity.py

```
    @classmethod
    def getcolumnvalues(cls,column):
           if not column in cls.columns :
                  raise KeyError('unknown column '+column)
           sql="select %s from %s order by lower(%s)"%(column,
                  cls.__name__,column)
           cursor=cls.threadlocal.connection.cursor()
           cursor.execute(sql)
           return [r[0] for r in cursor.fetchall()]
```

`getcolumnvalues()` first checks if the requested column exists in this `Entity` (sub) class and raises an exception if it doesn't. Then it constructs a SQL statement to return the values in the requested column, sorted without regard for case (highlighted). The result is a list of tuples consisting of a single item and this is converted to a simple list of items before returning it.

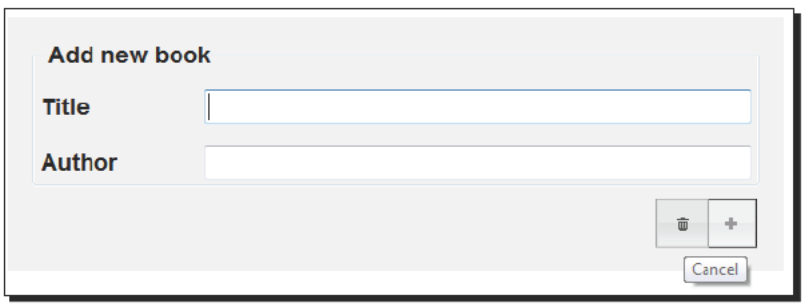
The presentation layer

Now that we have implemented the delivery layer, the application is almost usable, but looks a bit rough on the edges. Although some components already look quite good due to the styling inherent in the jQuery UI widgets used, other parts need some serious tweaking.

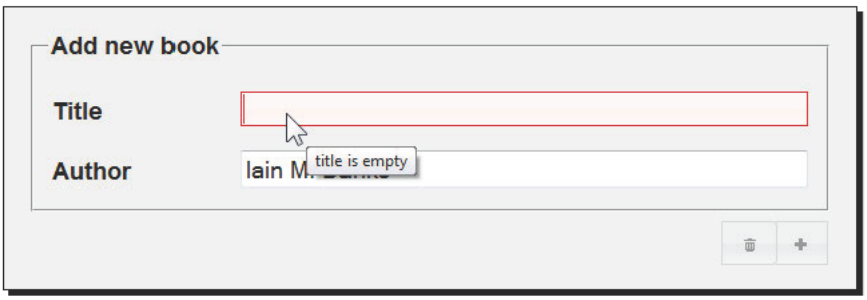
Time for action – using an enhanced presentation layer

The additional JavaScript code and CSS information is part of `booksweb.js` and `booksweb.css` respectively. The following illustrations show the end results for the page with the list of books and the page to add a new book:

We added some zebra stripes to aid readability and changed the look of the column headings.



The page to add a book had its buttons styled in the same style as the buttons on the page with the list of books. Also, the layout was cleaned up and functionality was added to present any errors returned in a clearly visible way (in the last example, the title is empty so the background is red).



What just happened?

To effect the changes seen in the previous images, we added the following lines of JavaScript to `booksweb.js`:

Chapter5/booksweb.js

```
$(".buttonbar").buttonset();
$("#newbook button[name=submit]").button({
                text: false,
                icons: {
                        primary: "ui-icon-plusthick"
                }
});
$("#newbook button[name=cancel]").button({
                text: false,
                icons: {
                        primary: "ui-icon-trash"
                }
});
```

The effect is just to alter the appearance of the buttons, not to add to their functionality with some sort of event handler because there is no need for this. The page contains a regular `<form>` element with a valid `action` attribute, so our submit and cancel buttons will behave as expected.

The rest of the changes, including borders, fonts, and colors are implemented in `booksweb.css`, which we do not examine here as the CSS contained in it is very straightforward.

Summary

We have learned a lot in this chapter about designing and implementing a web application around a data model consisting of several entities and relations.

Specifically, we covered:

- How to design the data model
- How to create a reusable entity and relation framework
- How to maintain a clear separation between database, object layer, and delivery layer
- How to implement auto completion using jQuery UI's autocomplete widget

We also discussed the importance of input validation, both client-side and server-side.

We did not yet wield the full power of our entity and relation framework and input validation might be much more involved. To exercise our new skills and expand them, the next chapter will be about designing and building a wiki application.

6
Building a Wiki

Nowadays, a wiki is a well-known tool to enable people to maintain a body of knowledge in a cooperative way. Wikipedia (`http://wikipedia.org`*) might be the most famous example of a wiki today, but countless numbers of forums use some sort of wiki and many tools and libraries exist to implement a wiki application.*

In this chapter, we will develop a wiki of our own, and in doing so, we will focus on two important concepts in building web applications. The first one is the design of the data layer. We will build upon the simple framework created in the previous chapter and we will try to establish where the limitations in our current implementation lie. The wiki application we will be building is a good test case as it is considerably more complex than the book database developed earlier.

The second one is input validation. A wiki is normally a very public application that might not even employ a basic authentication scheme to identify users. This makes contributing to a wiki very simple, yet also makes a wiki vulnerable in the sense that anyone can put anything on a wiki page. It's therefore a good idea to verify the content of any submitted change. You may, for example, strip out any HTML markup or disallow external links.

Enhancing user interactions in a meaningful way is often closely related with input validation. As we saw in the previous chapter, client-side input validation helps prevent the user from entering unwanted input and is therefore a valuable addition to any application but is not a substitute for server-side input validation as we cannot trust the outside world not to try and access our server in unintended ways.

We will address both input validation and user interaction explicitly when we develop our wiki application in this chapter.

In this chapter, we will:

- Implement a data layer for a wiki application
- Implement a delivery layer
- Take a good look at input validation
- Encounter jQuery UI's dialog widget

So let's get on with it...

The data layer

A wiki consists of quite a number of distinct entities we can indentify. We will implement these entities and the relations that exist between them by reusing the Entity/Relation framework developed earlier.

Time for action – designing the wiki data model

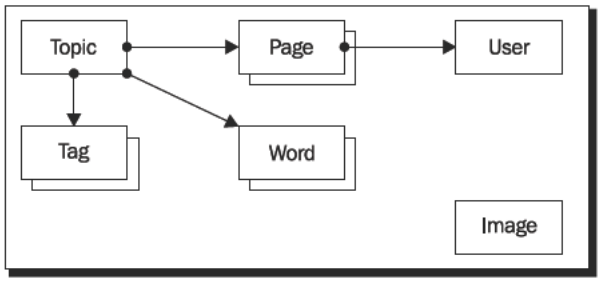
As with any application, when we start developing our wiki application we must first take a few steps to create a data model that can act as a starting point for the development:

1. Identify each entity that plays a role in the application. This might depend on the requirements. For example, because we want the user to be able to change the title of a topic and we want to archive revisions of the content, we define separate Topic and Page entities.

2. Identify direct relations between entities. Our decision to define separate Topic and Page entities implies a relation between them, but there are more relations that can be identified, for example, between Topic and Tag. Do not specify indirect relations: All topics marked with the same tag are in a sense related, but in general, it is not necessary to record these indirect relations as they can easily be inferred from the recorded relation between topics and tags.

The image shows the different entities and relations we can identify in our wiki application. Note that like in the books application, a User is a separate entity that is distinct from any user in, for example, a password database.

In the diagram, we have illustrated the fact that a `Topic` may have more than one `Page` while a `Page` refers to a single User in a rather informal way by representing Page as a stack of rectangles and `User` as a single rectangle. In this manner, we can grasp the most relevant aspects of the relations at a glance. When we want to show more relations or relations with different characteristics, it might be a good idea to use more formal methods and tools. A good starting point is the Wikipedia entry on UML: `http://en.wikipedia.org/wiki/Unified_Modelling_Language`.



What just happened?

With the entities and relations in our data model identified, we can have a look at their specific qualities.

The basic entity in a wiki is a `Topic`. A topic, in this context, is basically a title that describes what this topic is about. A topic has any number of associated Pages. Each instance of a `Page` represents a revision; the most recent revision is the *current* version of a topic. Each time a topic is edited, a new revision is stored in the database. This way, we can simply revert to an earlier version if we made a mistake or compare the contents of two revisions. To simplify identifying revisions, each revision has a modification date. We also maintain a relation between the `Page` and the `User` that modified that `Page`.

In the wiki application that we will develop, it is also possible to associate any number of tags with a topic. A `Tag` entity consists simply of a `tag` attribute. The important part is the relation that exists between the `Topic` entity and the `Tag` entity.

Like a `Tag`, a `Word` entity consists of a single attribute. Again, the important bit is the relation, this time, between a `Topic` and any number of `Words`. We will maintain this relation to reflect the words used in the current versions (that is, the last revision of a `Page`) of a `Topic`. This will allow for fairly responsive full text search facilities.

The final entity we encounter is the `Image` entity. We will use this to store images alongside the pages with text. We do not define any relation between topics and images. Images might be referred to in the text of the topic, but besides this textual reference, we do not maintain a formal relation. If we would like to maintain such a relation, we would be forced to scan for image references each time a new revision of a page was stored, and probably we would need to signal something if a reference attempt was made to a non-existing image. In this case, we choose to ignore this: references to images that do not exist in the database will simply show nothing:

Chapter6/wikidb.py

```
from entity import Entity
from relation import Relation

class User(Entity): pass
```

```
class Topic(Entity): pass
class Page(Entity): pass
class Tag(Entity): pass
class Word(Entity): pass
class Image(Entity): pass

class UserPage(Relation): pass
class TopicPage(Relation): pass
class TopicTag(Relation): pass
class ImagePage(Relation): pass
class TopicWord(Relation): pass

def threadinit(db):
    User.threadinit(db)
    Topic.threadinit(db)
    Page.threadinit(db)
    Tag.threadinit(db)
    Word.threadinit(db)
    Image.threadinit(db)
    UserPage.threadinit(db)
    TopicPage.threadinit(db)
    TopicTag.threadinit(db)
    ImagePage.threadinit(db)
    TopicWord.threadinit(db)

def inittable():
    User.inittable(userid="unique not null")
    Topic.inittable(title="unique not null")
    Page.inittable(content="",
                   modified="not null default CURRENT_TIMESTAMP")
    Tag.inittable(tag="unique not null")
    Word.inittable(word="unique not null")
    Image.inittable(type="",data="blob",title="",
                   modified="not null default CURRENT_TIMESTAMP",
                   description="")

    UserPage.inittable(User,Page)
    TopicPage.inittable(Topic,Page)
    TopicTag.inittable(Topic,Tag)
    TopicWord.inittable(Topic,Word)
```

Because we can reuse the entity and relation modules we developed earlier, the actual implementation of the database layer is straightforward (full code is available as `wikidb.py`). After importing both modules, we first define a subclass of `Entity` for each entity we identified in our data model. All these classes are used as is, so they have only a `pass` statement as their body.

Likewise, we define a subclass of `Relation` for each relation we need to implement in our wiki application.

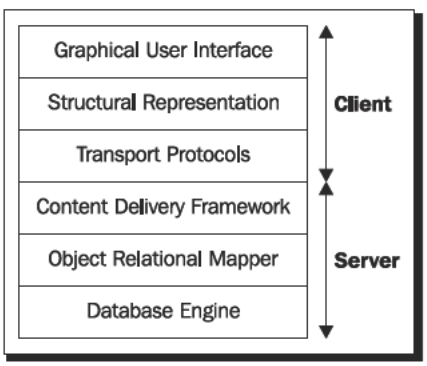
All these `Entity` and `Relation` subclasses still need the initialization code to be called once each time the application starts and that is where the convenience function `initdb()` comes in. It bundles the initialization code for each entity and relation (highlighted).

Many entities we define here are simple but a few warrant a closer inspection. The `Page` entity contains a `modified` column that has a `non null` constraint. It also has a default: `CURRENT_TIMESTAMP` (highlighted). This default is SQLite specific (other database engines will have other ways of specifying such a default) and will initialize the modified column to the current date and time if we create a new `Page` record without explicitly setting a value.

The `Image` entity also has a definition that is a little bit different: its `data` column is explicitly defined to have a `blob` affinity. This will enable us to store binary data without any problem in this table, something we need to store and retrieve the binary data contained in an image. Of course, SQLite will happily store anything we pass it in this column, but if we pass it an array of bytes (not a string that is), that array is stored as is.

The delivery layer

With the foundation, that is, the data layer in place, we build on it when we develop the delivery layer. Between the delivery layer and the database layer, there is an additional layer that encapsulates the domain-specific knowledge (that is, it knows how to verify that the title of a new `Topic` entity conforms to the requirements we set for it before it stores it in the database):



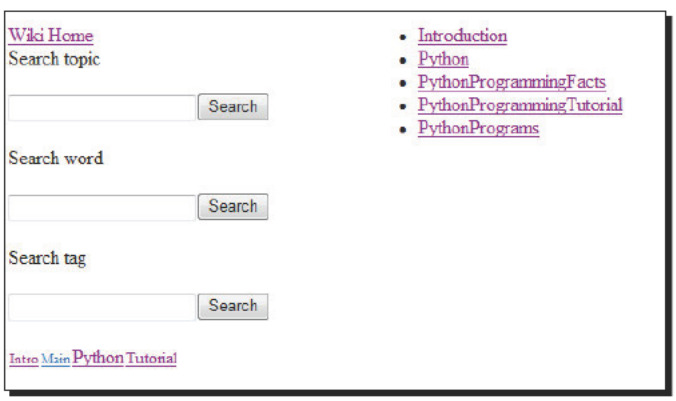
Each different layer in our application is implemented in its own file or files. It is easy to get confused, so before we delve further into these files, have a look at the following table. It lists the different files that together make up the wiki application and refers to the names of the layers introduced in *Chapter 1*, *Choosing Your Tools* (shown again in the previous image).

File	Layer	
`wikiweb.py`	Content Delivery Framework	Our main CherryPy application
`wiki.py`	Object Relational Mapper	The domain specific part; Imported by `wikiweb.py`
`wikidb.py`	Object Relational Mapper	The domain independent part; Imported by `wikiweb.py`
`basepage.html`	Structural Representation	Used by `wikiweb.py` to serve pages to the client
`wikiweb.js`	Graphical User Interface	Referred to in `basepage.html`; Implements user interaction like mouse clicks.
`wiki.css`	Graphical User Interface	Referred to in `basepage.html`; Implements the layout of graphical components.

We'll focus on the main CherryPy application first to get a feel for the behavior of the application.

Time for action – implementing the opening screen

The opening screen of the wiki application shows a list of all defined topics on the right and several ways to locate topics on the left. Note that it still looks quite rough because, at this point, we haven't applied any style sheets:



Let us first take a few steps to identify the underlying structure. This structure is what we would like to represent in the HTML markup:

- Identify related pieces of information that are grouped together. These form the backbone of a structured web page. In this case, the search features on the left form a group of elements distinct from the list of topics on the right.
- Identify distinct pieces of functionality within these larger groups. For example, the elements (input field and search button) that together make up the word search are such a piece of functionality, as are the tag search and the tag cloud.
- Try to identify any hidden functionality, that is, necessary pieces of information that will have to be part of the HTML markup, but are not directly visible on a page. In our case, we have links to the jQuery and JQuery UI JavaScript libraries and links to CSS style sheets.

Identifying these distinct pieces will not only help to put together HTML markup that reflects the structure of a page, but also help to identify necessary functionality in the delivery layer because each of these functional pieces is concerned with specific information processed and produced by the server.

What just happened?

Let us look in somewhat more detail at the structure of the opening page that we identified.

Most notable are three search input fields to locate topics based on words occurring in their bodies, based on their actual title or based on tags associated with a topic. These search fields feature auto complete functionality that allows for comma-separated lists. In the same column, there is also room for a tag cloud, an alphabetical list of tags with font sizes dependent on the number of topics marked with that tag.

The structural components

The HTML markup for this opening page is shown next. It is available as the file `basepage.html` and the contents of this file are served by several methods in the `Wiki` class implementing the delivery layer, each with a suitable content segment. Also, some of the content will be filled in by AJAX calls, as we will see in a moment:

Chapter6/basepage.html

```
<html>
   <head>
          <title>Wiki</title>
          <script
                 src=
"http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"
```

```
                type="text/javascript">
            </script>
            <script
                src=
"http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.3/jquery-ui.min.js"
                type="text/javascript">
            </script>
            <link rel="stylesheet"
                href="http://ajax.googleapis.com/ajax/libs/
jqueryui/1.8.3/themes/smoothness/jquery-ui.css"
                type="text/css" media="all" />
            <link rel="stylesheet" href="/wiki.css"
                type="text/css" media="all" />
    </head>
    <body>
            <div id="navigation">
                <div class="navitem">
                    <a href="./">Wiki Home</a>
                </div>
                <div class="navitem">
                    <span class="label">Search topic</span>
                    <form id="topicsearch">
                        <input type="text" >
                        <button type="submit" >Search</button>
                    </form>
                </div>
                <div class="navitem">
                    <span class="label">Search word</span>
                    <form id="wordsearch">
                        <input type="text" >
                        <button type="submit" >Search</button>
                    </form>
                </div>
                <div class="navitem">
                    <span class="label">Search tag</span>
                    <form id="tagsearch">
                        <input type="text" >
                        <button type="submit" >Search</button>
                    </form>
                </div>
                <div class="navitem">
                    <p id="tagcloud">Tag cloud</p>
                </div>
            </div>
```

```
            <div id="content">%s</div>
            <script src="/wikiweb.js" type="text/javascript"></script>
        </body>
    </html>
```

The `<head>` element contains both links to CSS style sheets and `<script>` elements that refer to the jQuery libraries. This time, we choose again to retrieve these libraries from a public content delivery network.

The highlighted lines show the top-level `<div>` elements that define the structure of the page. In this case, we have identified a navigation part and a content part and this is reflected in the HTML markup.

Enclosed in the navigation part are the search functions, each in their own `<div>` element. The content part contains just an interpolation placeholder `%s` for now, that will be filled in by the method that serves this markup. Just before the end of the body of the markup is a final `<script>` element that refers to a JavaScript file that will perform actions specific to our application and we will examine those later.

The application methods

The markup from the previous section is served by methods of the `Wiki` class, an instance of which class can be mounted as a CherryPy application. The `index()` method, for example, is where we produce the markup for the opening screen (the complete file is available as `wikiweb.py` and contains several other methods that we will examine in the following sections):

Chapter6/wikiweb.py

```
@cherrypy.expose
def index(self):
        item = '<li><a href="show?topic=%s">%s</a></li>'
        topiclist = "\n".join(
                [item%(t,t)for t in wiki.gettopiclist()])
        content = '<div id="wikihome"><ul>%s</ul></div>'%(
                topiclist,)
        return basepage % content
```

First, we define the markup for every topic we will display in the main area of the opening page (highlighted). The markup consists of a list item that contains an anchor element that refers to a URL relative to the page showing the opening screen. Using relative URLs allows us to mount the class that implements this part of the application anywhere in the tree that serves the CherryPy application. The `show()` method that will serve this URL takes a topic parameter whose value is interpolated in the next line for each topic that is present in the database.

The result is joined to a single string that is interpolated into yet another string that encapsulates all the list items we just generated in an unordered list (a `<ul>` element in the markup) and this is finally returned as the interpolated content of the `basepage` variable.

In the definition of the `index()` method, we see a pattern that will be repeated often in the wiki application: methods in the delivery layer, like `index()`, concern themselves with constructing and serving markup to the client and delegate the actual retrieval of information to a module that knows all about the wiki itself. Here the list of topics is produced by the `wiki.gettopiclist()` function, while `index()` converts this information to markup. Separation of these activities helps to keep the code readable and therefore maintainable.

Time for action – implementing a wiki topic screen

When we request a URL of the form `show?topic=value`, this will result in calling the `show()` method. If `value` equals an existing topic, the following (as yet unstyled) screen is the result:

Just as for the opening screen, we take steps to:

- Identify the main areas on screen
- Identify specific functionality
- Identify any hidden functionality

The page structure is very similar to the opening screen, with the same navigational items, but instead of a list of topics, we see the content of the requested topic together with some additional information like the tags associated with this subject and a button that may be clicked to edit the contents of this topic. After all, collaboratively editing content is what a Wiki is all about.

We deliberately made the choice not to refresh the contents of just a part of the opening screen with an AJAX call, but opted instead for a simple link that replaces the whole page. This way, there will be an unambiguous URL in the address bar of the browser that will point at the topic. This allows for easy bookmarking. An AJAX call would have left the URL of the opening screen that is visible in the address bar of the browser unaltered and although there are ways to alleviate this problem, we settle for this simple solution here.

What just happened?

As the main structure we identified is almost identical to the one for the opening page, the `show()` method will reuse the markup in `basepage.html`.

Chapter6/wikiweb.py

```
@cherrypy.expose
def show(self,topic):
       topic = topic.capitalize()
       currentcontent,tags = wiki.gettopic(topic)
       currentcontent = "".join(wiki.render(currentcontent))
       tags = ['<li><a href="searchtags?tags=%s">%s</a></li>'%(
                          t,t) for t in tags]
       content = '''
       <div>
              <h1>%s</h1><a href="edit?topic=%s">Edit</a>
       </div>
       <div id="wikitopic">%s</div>
       <div id="wikitags"><ul>%s</ul></div>
       <div id="revisions">revisions</div>
       ''' % ( topic, topic, currentcontent,"\n".join(tags))
       return basepage % content
```

The `show()` method delegates most of the work to the `wiki.gettopic()` method (highlighted) that we will examine in the next section and concentrates on creating the markup it will deliver to the client. `wiki.gettopic()` will return a tuple that consists of both the current content of the topic and a list of tags.

Those tags are converted to `<li>` elements with anchors that point to the `searchtags` URL. This list of tags provides a simple way for the reader to find related topics with a single click. The `searchtags` URL takes a `tags` argument so a single `<li>` element constructed this way may look like this: `<li><a href="searchtags?tags=Python">Python</a></li>`.

The content and the clickable list of tags are embedded in the markup of the `basepage` together with an anchor that points to the `edit` URL. Later, we will style this anchor to look like a button and when the user clicks it, it will present a page where the content may be edited.

Time for action – editing wiki topics

In the previous section, we showed how to present the user with the contents of a topic but a wiki is not just about finding topics, but must present the user with a way to edit the content as well. This edit screen is presented in the following screenshot:

Besides the navigation column on the left, within the edit area, we can point out the following functionality:

- Elements to alter the title of the subject.
- Modify the tags (if any) associated with the topic.
- A large text area to edit the contents of the topic. On the top of the text area, we see a number of buttons that can be used to insert references to other topics, external links, and images.
- A **Save** button that will submit the changes to the server.

What just happened?

The `edit()` method in `wikiweb.py` is responsible for showing the edit screen as well as processing the information entered by the user, once the save button is clicked:

Chapter6/wikiweb.py

```
@cherrypy.expose
def edit(self,topic,
                    content=None,tags=None,originaltopic=None):
```

```
user = self.logon.checkauth(
       logonurl=self.logon.path, returntopage=True)
if content is None :
       currentcontent,tags = wiki.gettopic(topic)
       html = '''
       <div id="editarea">
              <form id="edittopic" action="edit"
                     method="GET">
                     <label for="topic"></label>
                     <input name="originaltopic"
                            type="hidden" value="%s">
                     <input name="topic" type="text"
                            value="%s">
                     <div id="buttonbar">
                            <button type="button"
                                   id="insertlink">
                                   External link
                            </button>
                            <button type="button"
                                   id="inserttopic">
                                   Wiki page
                            </button>
                            <button type="button"
                                   id="insertimage">
                                   Image
                            </button>
                     </div>
                     <label for="content"></label>
                     <textarea name="content"
                            cols="72" rows="24" >
                            %s
                     </textarea>
                     <label for="tags"></label>
                     <input name="tags" type="text"
                            value="%s">
                     <button type="submit">Save</button>
                     <button type="button">Cancel</button>
                     <button type="button">Preview</button>
              </form>
       </div>
       <div id="previewarea">preview</div>
       <div id="imagedialog">%s</div>
       <script>
              $("#imagedialog").dialog(
```

```
                                {autoOpen:false,
                                 width:600,
                                 height:600});
            </script>
            '''%(topic, topic, currentcontent,
                                ", ".join(tags),
                                "".join(self.images()))
            return basepage % html
    else :
            wiki.updatetopic(originaltopic,topic,content,tags)
            raise cherrypy.HTTPRedirect('show?topic='+topic)
```

The first priority of the `edit()` method is to verify that the user is logged in as we want only known users to edit the topics. By setting the `returntopage` parameter to true, the `checkauth()` method will return to this page once the user is authenticated.

The `edit()` method is designed to present the edit screen for a topic as well as to process the result of this editing when the user clicks the **Save** button and therefore takes quite a number of parameters.

The distinction is made based on the `content` parameter. If this parameter is not present (highlighted), the method will produce the markup to show the various elements in the edit screen. If the content parameter is not equal to `None`, the `edit()` method was called as a result of submitting the content of the form presented in the edit screen, in which case, we delegate the actual update of the content to the `wiki.updatetopic()` method. Finally, we redirect the client to a URL that will show the edited content again in its final form without the editing tools.

At this point, you may wonder what all this business is about with both a `topic` and an `originaltopic` parameter. In order to allow the user to change the title of the topic while that title is also used to find the topic entity that we are editing, we pass the title of the topic as a hidden variable in the edit form, and use this value to retrieve the original topic entity, a ploy necessary because, at this point, we may have a new title and yet have to find the associated topic that still resides in the database with the old title.

Cross Site Request Forgery

When we process the data sent to the `edit()` function we make sure that only authenticated users submit anything. Unfortunately, this might not be enough if the user is tricked into sending an authenticated request on behalf of someone else. This is called **Cross Site Request Forgery** (**CSRF**) and although there are ways to prevent this, these methods are out of scope for this example. Security conscious people should read up on these exploits, however, and a good place to start is `http://www.owasp.org/index.php/Main_Page` and for Python-specific discussions `http://www.pythonsecurity.org/`.

Pop quiz

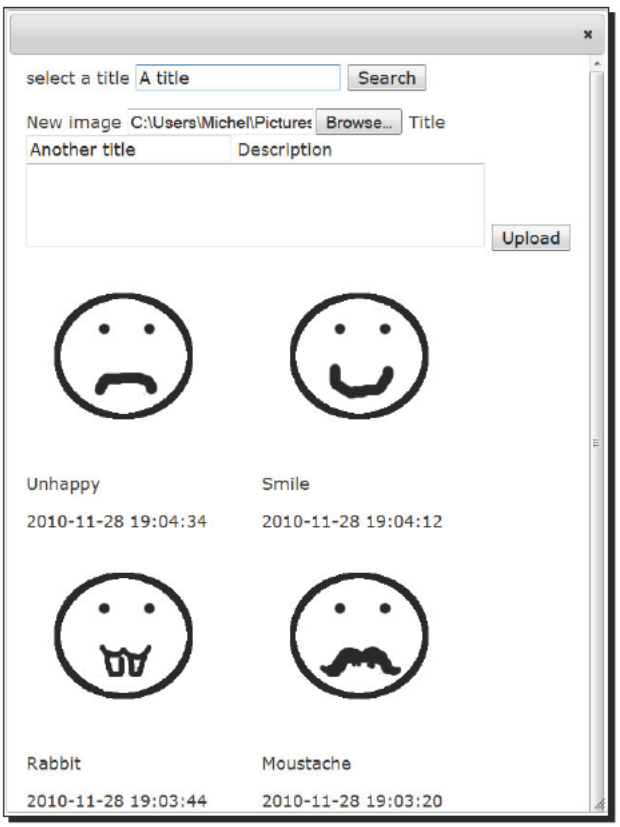
What other attribute of the `Topic` entity could we have passed to retrieve a reference to the topic we are editing?

Additional functionality

In the opening screen as well as in the pages showing the content of topics and in the editing page, there is a lot of hidden functionality. We already encountered several functions of the `wiki` module and we will examine them in detail in this section together with some JavaScript functionality to enhance the user interface.

Time for action – selecting an image

On the page that allows us to edit a topic, we have half hidden an important element: the dialog to insert an image. If the insert image button is clicked, a dialog is present, as shown in the following image:



Because a dialog is, in a way, a page of its own, we take the same steps to identify the functional components:

- Identify the main structure
- Identify specific functional components
- Identify hidden functionality

The dialog consists of two forms. The top one consists of an input field that can be used to look for images with a given title. It will be augmented with jQuery UI's auto complete functionality.

The second form gives the user the possibility to upload a new file while the rest of the dialog is filled with any number of images. Clicking on one of the images will close the dialog and insert a reference to that image in the text area of the edit page. It is also possible to close the dialog again without selecting an image by either clicking the small close button on the top-right or by pressing the *Escape* key.

What just happened ?

The whole dialog consists of markup that is served by the `images()` method.

Chapter6/wikiweb.py

```
@cherrypy.expose
def images(self,title=None,description=None,file=None):
       if not file is None:
              data = file.file.read()
              wikidb.Image(title=title,description=description,
                     data=data,type=str(file.content_type))
       yield '''
       <div>
              <form>
                     <label for="title">select a title</label>
                     <input name="title" type="text">
                     <button type="submit">Search</button>
              </form>
              <form method="post" action="./images"
                     enctype="multipart/form-data">
                     <label for="file">New image</label>
                     <input type="file" name="file">
                     <label for="title">Title</label>
                     <input type="text" name="title">
                     <label for="description">Description</label>
                     <textarea name="description"
                            cols="48" rows="3"></textarea>
                     <button type="submit">Upload</button>
```

```
        </form>
</div>
'''
yield '<div id="imagelist">\n'
for img in self.getimages():
        yield img
yield '</div>'
```

There is some trickiness here to understand well: from the `edit()` method, we call this `images()` method to provide the markup that we insert in the page that is delivered to the client requesting the `edit` URL, but because we have decorated the `images()` method with a `@cherrypy.expose` decorator, the `images()` method is visible from the outside and may be requested with the `images` URL. If accessed that way, CherryPy will take care of adding the correct response headers.

Being able to call this method this way is useful in two ways: because the dialog is quite a complex page with many elements, we may check how it looks without being bothered by it being part of a dialog, and we can use it as the target of the form that is part of the images dialog and that allows us to upload new images. As with the `edit()` method, the distinction is again made based on a whether a certain parameter is present. The parameter that serves this purpose is `file` and will contain a `file` object if this method is called in response to an image being submitted (highlighted).

The `file` object is a `cherrypy.file` object, not a Python built in `file` object, and has several attributes, including an attribute called `file` that is a regular Python stream object. This Python stream object serves as an interface to a temporary file that CherryPy has created to store the uploaded file. We can use the streams `read()` method to get at its content.

Sorry about all the references to `file`, I agree it is possibly a bit confusing. Read it twice if needed and relax. This summary may be convenient:

This item	has a	which is a
The `images()` method	`file` parameter	`herrypy.file` object
A `cherrypy.file` object	`file` attribute	Python `stream` object
A Python `stream` object	`name` attribute	name of a file on disk

The Python stream can belong to a number of classes where all implement the same API. Refer to `http://docs.python.org/py3k/library/functions.html#open` for details on Python streams.

The `cherrypy.file` also has a `content_type` attribute whose string representation we use together with the title and the binary data to create a new `Image` instance.

The next step is to present the HTML markup that will produce the dialog, possibly including the uploaded image. This markup contains two forms.

The first one (highlighted in the previous code snippet) consists of an input field and a submit button. The input field will be augmented with auto complete functionality as we will see when we examine `wikiweb.js`. The submit button will replace the selection of images when clicked. This is also implemented in `wikiweb.js` by adding a click handler that will perform an AJAX call to the `getimages` URL.

The next form is the file upload form. What makes it a file upload form is the `<input>` element of the type `file` (highlighted). Behind the scenes, CherryPy will store the contents of a file type `<input>` element in a temporary file and pass it to the method servicing the requested URL by submitting the form.

There is a final bit of magic to pay attention to: we insert the markup for the dialog as part of the markup that is served by the `edit()` method, yet the dialog only shows if the user clicks the insert image button. This magic is performed by jQuery UI's dialog widget and we convert the `<div>` element containing the dialog's markup by calling its `dialog` method, as shown in this snippet of markup served by the `edit()` method():

```
<script>$("#imagedialog").dialog({autoOpen:false});</script>
```

By setting the `autoOpen` option to false, we ensure that the dialog remains hidden when the page is loaded, after all, the dialog should only be opened if the user clicks the insert image button.

Opening the dialog is accomplished by several pieces of JavaScript (full code available as `wikiweb.js`). The first piece associates a click handler with the insert image button that will pass the `open` option to the dialog, causing it to display itself:

Chapter6/wikiweb.js

```
$("#insertimage").click(function(){
    $("#imagedialog").dialog("open");
});
```

Note that the default action of a dialog is to close itself when the *Escape* key is pressed, so we don't have to do anything about that.

Within the dialog, we have to configure the images displayed there to insert a reference in the text area when clicked and then close the dialog. We do this by configuring a `live` handler for the `click` event. A `live` handler will apply to elements that match the selector (in this case, images with the `selectable-image` class) even if they are not present yet. This is crucial, as we may upload new images that are not yet present in the list of images shown when the dialog is first loaded:

Chapter6/wikiweb.js

```
$(".selectable-image").live('click',function(){
    $("#imagedialog").dialog("close");
```

```
    var insert = "<" + $(this).attr("id").substring(3) + "," +
$(this).attr("alt") + ">";
    var Area = $("#edittopic textarea");
    var area = Area[0];
    var oldposition = Area.getCursorPosition();

var pre  = area.value.substring(0, oldposition);
    var post = area.value.substring(oldposition);
    area.value = pre + insert + post;
    Area.focus().setCursorPosition(oldposition + insert.length);
});
```

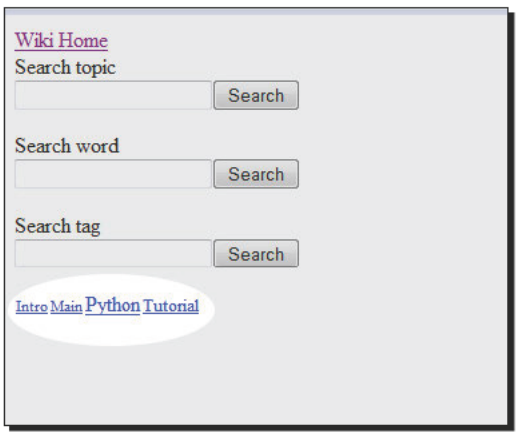
The first activity of this handler is to close the dialog. The next step is to determine what text we would like to insert into the text area (highlighted). In this case, we have decided to represent a reference to an image within the database as a number followed by a description within angled brackets. For example, image number 42 in the database might be represented as `<42,"Picture of a shovel">`. When we examine the `render()` method in `wikiweb.py`, we will see how we will convert this angled bracket notation to HTML markup.

The remaining part of the function is concerned with inserting this reference into the `<textarea>` element. We therefore retrieve the jQuery object that matches our text area first (highlighted) and because such a selection is always an array and we need access to the underlying JavaScript functionality of the `<textarea>` element, we fetch the first element.

The `value` attribute of a `<textarea>` element holds the text that is being edited and we split this text into a part before the cursor position and a part after it and then combine it again with our image reference inserted. We then make sure the text area has the focus again (which might have shifted when the user was using the dialog) and position the cursor at a position that is just after the newly inserted text.

Time for action – implementing a tag cloud

One of the distinct pieces of functionality we identified earlier was a so called tag cloud.



The tag cloud that is present in the navigation section of all pages shows an alphabetically sorted list of tags. The styling of the individual tags represents the relative number of topics that are marked with this tag. Clicking on the tags will show the list of associated topics. In this implementation, we vary just the font size but we could have opted for additional impact by varying the color as well.

Before we implement a tag cloud, we should take a step back and take a good look at what we need to implement:

- We need to retrieve a list of tags
- We need to sort them
- We need to present markup. This markup should contain links that will refer to a suitable URL that will represent a list of topics that are marked with this tag. Also, this markup must in some way indicate what the relative number of topics is that have this tag so it can be styled appropriately.

The last requirement is again a matter of separating structure from representation. It is easier to adapt a specific style by changing a style sheet than to alter structural markup.

What just happened?

If we look at the HTML that represents an example tag cloud, we notice that the tags are represented by `<span>` elements with a `class` attribute that indicates its weight. In this case, we divide the range of weights in five parts, giving us classes from `weight0` for the least important tag to `weight4` for the most important one:

```
<span class="weight1"><a href="searchtags?tags=Intro">Intro</a></span>
<span class="weight1"><a href="searchtags?tags=Main">Main</a></span>
<span class="weight4"><a href="searchtags?tags=Python">Python</a></
span>
<span class="weight2"><a href="searchtags?tags=Tutorial">Tutorial</
a></span>
```

The actual font size we use to represent these weights is determined by the styles in `wiki.css`:

```
.weight0 { font-size:60%; }
.weight1 { font-size:70%; }
.weight2 { font-size:80%; }
.weight3 { font-size:90%; }
.weight4 { font-size:100%; }
```

The tag cloud itself is delivered by the `tagcloud()` method in `wikiweb.py`.

Chapter6/wikiweb.py

```
@cherrypy.expose
def tagcloud(self,_=None):
      for tag,weight in wiki.tagcloud():
            yield '''
            <span class="weight%s">
                  <a href="searchtags?tags=%s">%s</a>
            </span>'''%(weight,tag,tag)
```

This method iterates over all tuples retrieved from `wiki.tagcloud()` (highlighted). These tuples consist of a weight and a tag name and these are transformed to links and encapsulated in a `<span>` element with a fitting `class` attribute:

Chapter6/wiki.py

```
def tagcloud():
     tags = sorted([wikidb.Tag(id=t) for t in wikidb.Tag.list()],
                        key=attrgetter('tag'))
     totaltopics=0
     tagrank = []
     for t in tags:
          topics = wikidb.TopicTag.list(t)
          if len(topics):
               totaltopics += len(topics)
               tagrank.append((t.tag,len(topics)))
     maxtopics = max(topics for tag,topics in tagrank)
     for tag,topics in tagrank:
          yield tag, int(5.0*topics/(maxtopics+1)) # map to 0 - 4
```

The `tagcloud()` function in `wiki.py` starts off by retrieving a list of all `Tag` objects and sorts them based on their `tag` attribute. Next, it iterates over all these tags and retrieves their associated topics (highlighted). It then checks if there really are topics by checking the length of the list of topics. Some tags may not have any associated topics and are not counted in this ranking operation.

When a tag is removed from a topic, we do not actually delete the tag itself if it no longer has any associated topics. This might lead to a buildup of unused tags and, if necessary, you might want to implement some clean-up scheme.

If a tag does have associated topics, the number of topics is added to the total and a tuple consisting of the tag name and the number of topics is appended to the `tagrank` list. Because our list of `Tag` objects was sorted, `tagrank` will be sorted as well when we have finished counting the topics.

In order to determine the relative weight of the tags, we iterate again, this time over the `tagrank` list to find the maximum number of topics associated with any tag. Then, in a final iteration, we yield a tuple consisting of the tag name and it relative weight, where the relative weight is computed by dividing the number of topics by the maximum number we encountered (plus one, to prevent divide by zero errors). This weight will then be between zero and one (exclusive) and by multiplying this by 5 and rounding down to an integer, a whole number between 0 and 4 (inclusive) is obtained.

Time for action – searching for words

To be able to find a list of all topics which contain one or more specific words, we present the user with a search form in the navigation area. These are some of the considerations when designing such a form:

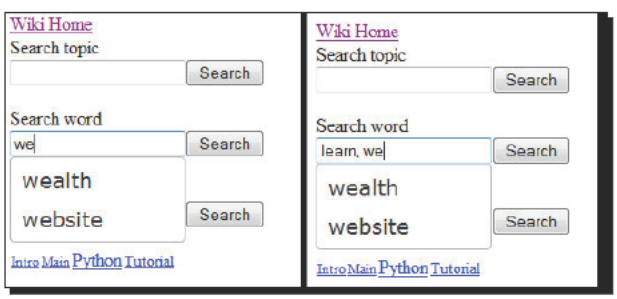
- The user must be able to enter more than one word to find topics with all those words in their content
- Searching should be case insensitive
- Locating those topics should be fast even if we have a large number of topics with lots of text
- Auto completion would be helpful to aid the user in specifying words that are actually part of the content of some topic

All these considerations will determine how we will implement the functionality in the delivery layer and on the presentation side.

What just happened?

The search options in the navigation area and the tag entry field in the edit screen all feature autocomplete functionality. We encountered autocomplete functionality before in the previous chapter where it was employed to show a list of titles and authors.

With the word and tag search fields in the wiki application, we would like to go one step further. Here we would like to have auto completion on the list of items separated by commas. The illustrations show what happens if we type a single word and what happens when a second word is typed in:



We cannot simply send the list of items complete with commas to the server because in that case we could not impose a minimum character limit. It would work for the first word of course, but once the first word is present in the input field, each subsequent character entry would result in a request to the server whereas we would like this to happen when the minimum character count for the second word is reached.

Fortunately, the jQuery UI website already shows an example of how to use the autocomplete widget in exactly this situation (check the example at `http://jqueryui.com/demos/autocomplete/#multiple-remote`). As this online example is fairly well explained in its comments, we will not list it here, but note that the trick lies in the fact that instead of supplying the autocomplete widget with just a source URL, it is also given a callback function that will be invoked instead of retrieving information directly. This callback has access to the string of comma-separated items in the input field and can call the remote source with just the last item in the list.

On the delivery side, the word search functionality is represented by two methods. The first one is the `getwords()` method in `wikiweb.py`:

Chapter6/wikiweb.py

```
@cherrypy.expose
def getwords(self,term,_=None):
        term = term.lower()
        return json.dumps(
                [t for t in wikidb.Word.getcolumnvalues('word')
                        if t.startswith(term)])
```

`getwords()` will return a list of words that starts with the characters in the `term` argument and returns those as a JSON serialized string for use by the auto completion function that we will add to the input field of the word search form. Words are stored all lowercase in the database. Therefore, the `term` argument is lowercased as well before matching any words (highlighted). Note that the argument to `json.dumps()` is in square brackets to convert the generator returned by the list comprehension to a list. This is necessary because `json.dumps` does not accept generators.

The second method is called `searchwords()`, which will return a list of clickable items consisting of those topics that contain all words passed to it as a string of comma-separated words. The list will be alphabetically sorted on the name of the topic:

Chapter6/wikiweb.py

```
    @cherrypy.expose
    def searchwords(self,words):
           yield '<ul>\n'
           for topic in sorted(wiki.searchwords(words)):
                  yield '<li><a href="show?topic=%s">%s</a></li>'%(
                         topic,topic)
           yield '</ul>\n'
```

Note that the markup returned by `searchwords()` is not a complete HTML page, as it will be called asynchronously when the user clicks the search button and the result will replace the content part.

Again, the hard work of actually finding the topics that contain the words is not done in the delivery layer, but delegated to the function `wiki.searchwords()`:

Chapter6/wiki.py

```
def searchwords(words):
   topics = None
   for word in words.split(','):
          word = word.strip('.,:;!? ').lower() # a list with a final
comma will yield an empty last term
          if word.isalnum():
                 w = list(wikidb.Word.list(word=word))
                 if len(w):
                        ww = wikidb.Word(id=w[0])
                        wtopic = set( w.a_id for w in wikidb.
TopicWord.list(ww) )
                        if topics is None :
                               topics = wtopic
                        else:
                               topics &= wtopic
```

```
                        if len(topics) == 0 :
                              break
        if not topics is None:
               for t in topics:
                      yield wikidb.Topic(id=t).title
```

This `searchwords()` function starts by splitting the comma-separated items in its `word` argument and sanitizing each item by stripping, leading, and trailing punctuation and whitespace and converting it to lowercase (highlighted).

The next step is to consider only items that consist solely of alphanumeric characters because these are the only ones stored as word entities to prevent pollution by meaningless abbreviations or markup.

We then check whether the item is present in the database by calling the `list()` method of the `Word` class. This will return either an empty list or a list containing just a single ID. In the latter case, this ID is used to construct a `Word` instance and we use that to retrieve a list of `Topic` IDs associated with this word by calling the `list()` method of the `TopicWord` class (highlighted) and convert it to a set for easy manipulation.

If this is the first word we are checking, the `topics` variable will contain `None` and we simply assign the set to it. If the `topic` variable already contains a set, we replace the set by the intersection of the stored set and the set of topic IDs associated with the word we are now examining. The intersection of two sets is calculated by the `&` operator (in this case, replacing the left-hand side directly, hence the `&=` variant). The result of the intersection will be that we have a set of topic IDs of topics that contain all words examined so far.

If the resulting set contains any IDs at all, these are converted to `Topic` instances to yield their `title` attribute.

The importance of input validation

Anything that is passed as an argument to the methods that service the wiki application, can potentially damage the application. This may sound a bit pessimistic, but remember that when designing an application, you cannot rely on the goodwill of the public, especially when the application is accessible over the Internet and your public may consist of dimwitted search bots or worse.

We may limit the risks by granting the right to edit a page only to people we know by implementing some sort of authentication scheme, but we don't want even these people to mess up the appearance of a topic by inserting all sorts of HTML markup, references to images that do not exist or even malicious snippets of JavaScript. We therefore want to get rid of any unwanted HTML elements present in the content before we store it in the database, a process generally known as *scrubbing*.

Preventing Cross-Site Scripting (XSS) (as the inclusion of unwanted code in web pages is called) is covered in depth on this webpage: `http://www.pythonsecurity.org/wiki/cross-sitescripting/`.

Time for action – scrubbing your content

Many wikis do not allow any HTML markup at all, but use simpler markup methods to indicate bulleted lists, headers, and so on.

Check for examples of possible markup schemes, for example, *markdown* `http://daringfireball.net/projects/markdown/`, *REST* `http://docutils.sourceforge.net/rst.html`, or for markup that does allow some HTML–the *mediawiki* software at `http://en.wikipedia.org/wiki/MediaWiki`.

Consider the following:

- Will the user understand some HTML markup or opt for no HTML markup at all?
- What will the wiki contain? Just text or also external references or references to binary objects (like images) stored in the wiki?

For this wiki, we will implement a mixed approach. We will allow some HTML markup like `<b>` and `<ul>` but not any links. References to topics in the wiki might be entered as `[Topic]`, whereas links to external pages might be denoted as `{www.example.org}`. Images stored in the wiki may be referred to as `<143>`. Each type of reference will take an optional description as well. Example markup, as entered by the user, is shown next:

```
This topic is tried with a mix of legal and illegal markup.

A <b>list</b> is fine:
<ul>
<li>One</li>
<li>Two</li>
<li>Three</li>
</ul>

A link using an html tag referring to a <a href="http://www.example.
com" target="blank">nasty popup</a>.

A legal link uses braces {http://www.example.com, "A link"}
```

When viewed, it will look like the following image:

Illegalmarkup

Edit

This topic is tried with a mix of legal and illegal markup.

A **list** is fine:

- One
- Two
- Three

A link using an html tah not nasty popup.

A legal link uses braces "A link"

What just happened?

When we encountered the `edit()` method in `wikiweb.py`, we saw that the actual update of the content of a topic was delegated to the `updatetopic()` function in `wiki.py`, so let's have a look at how this function is organized:

Chapter6/wiki.py

```
def updatetopic(originaltopic,topic,content,tags):
    t=list(wikidb.Topic.list(title=originaltopic))
    if len(t) == 0 :
           t=wikidb.Topic(title=topic)
    else:
           t=wikidb.Topic(id=t[0])
           t.update(title=topic)
    content=scrub(content)
    p=wikidb.Page(content=content)
    wikidb.TopicPage(t.id,p.id)
    # update word index
    newwords = set(splitwords(content))
    wordlist = wikidb.TopicWord.list(t)
    topicwords = { wikidb.Word(id=w.b_id).word:w
                                for w in wordlist }
    updateitemrelation(t,topicwords,newwords,
           wikidb.Word,'word',wikidb.TopicWord)
    # update tags
    newtags = set(t.capitalize()
                                for t in [t.strip()
                                       for t in tags.split(',')] if
t.isalnum())
    taglist = wikidb.TopicTag.list(t)
```

```
        topictags = { wikidb.Tag(id=t.b_id).tag:t
                                  for t in taglist }
        updateitemrelation(t,topictags,newtags,
               wikidb.Tag,'tag',wikidb.TopicTag)
```

First it checks whether the topic already exists by retrieving a list of `Topic` objects that have a `title` attribute that matches the `originaltopic` parameter. If this list is empty, it creates a new topic (highlighted), otherwise we update the `title` attribute of the first matching topic found. (See the explanation of the `edit()` method for the rationale behind this).

Then it calls the `scrub()` function to sanitize the content and then creates a new `Page` instance to store this content and associates it with the `Topic` instance `t`. So every time we update the content, we create a new revision and old revisions are still available for comparison.

The next step is to update the list of words used in the topic. We therefore create a set of unique words by passing the content to the `splitwords()` function (not shown here, available in `wiki.py`) and converting the list of words to a set. Converting a list to a set will remove any duplicate items.

We convert the set of words to a dictionary with `Word` objects as keys and the words themselves as values and call the `updateitemrelation()` function to perform the update.

The same scenario is used with any tags associated with the topic. The `updateitemrelation()` function may look intimidating, but that is mainly due to the fact that it is made general enough to deal with any `Relation`, not just one between `Topic` and `Word` or `Topic` and `Tag`. By designing a general function, we have less code to maintain which is good although, in this case, readability may have suffered too much.

Chapter6/wiki.py

```
def updateitemrelation(p,itemmap,newitems,Entity,attr,Relation):
    olditems = set()
    for item in itemmap:
           if not item in newitems:
                  itemmap[item].delete()
           else:
                  olditems.add(item)
    for item in newitems - olditems:
           if not item in itemmap:
                  ilist = list(Entity.list(**{attr:item}))
                  if (len(ilist)):
                         i = Entity(id=ilist[0])
                  else:
                         i = Entity(**{attr:item})
                  Relation.add(p,i)
```

First we determine if any items currently associated with the primary entity `p` are not in the new list of items. If so, they are deleted, that is, the recorded relation between the primary entity and the item is removed from the database, otherwise we store them in the `olditems` set.

The next step determines the difference between the `newitems` and `olditmes` (highlighted). The result represents those items that have to be associated with the primary entity, but may not yet be stored in the database. This is determined by using the `list()` method to find any, and if no entity is found, to create one. Finally, we add a new relation between the primary entity and the item

The `scrub()` method is used to remove any HTML tags from the content that are not explicitly listed as being allowed:

Chapter6/wiki.py

```
def scrub(content):
   parser = Scrubber(('ul','ol','li','b','i','u','em','code','pre','h
1','h2','h3','h4'))
   parser.feed(content)
   return "".join(parser.result)
```

For this purpose, it instantiates a `Scrubber` object with a very limited list of allowable tags (highlighted) and feeds the content to its `feed()` method. The result is then found in the result attribute of the `Scrubber` instance:

Chapter6/wiki.py

```
class Scrubber(HTMLParser):
   def __init__(self,allowed_tags=[]):
          super().__init__()
          self.result = []
          self.allowed_tags = set(allowed_tags)

   def handle_starttag(self, tag, attrs):
          if tag in self.allowed_tags:
                 self.result.append('<%s %s>'%(tag,
                              " ".join('%s="%s"'%a for a in attrs)))

   def handle_endtag(self, tag):
          if tag in self.allowed_tags:
                 self.result.append('</'+tag+'>')

   def handle_data(self,data):
          self.result.append(data)
```

The `Scrubber` class is a subclass of the `HTMLParser` class provided in Python's `html.parser` module. We override suitable methods here to deal with start and end tags and data and ignore the rest (like processing instructions and the like). Both beginning and end tags are only appended to the result if they are present in the list of allowable tags. Regular data (text, that is) is simply appended to the result.

Time for action – rendering content

We added specific JavaScript functionality to the text area editor to insert references to external websites, other wiki topics, and wiki images in a format that we devised ourselves and that cannot be interpreted as HTML. Now we have to provide code that will convert this notation to something that will be understood by the client.

What just happened?

Recognizing those items that we have to convert to HTML is mostly done by using regular expressions. We therefore define three regular expressions first, each representing a distinct pattern. Note that we use raw strings here to prevent interpretation of backslashes. Backslashes are meaningful in regular expression, and if we didn't use raw strings, we would have to escape each and every backslash with a backslash, resulting in an unreadable sea of backslashes:

Chapter6/wiki.py

```
topicref = re.compile(r'\[\s*([^,\]]+?)(\s*,\s*([^\]]+))?\s*\]')
linkref  = re.compile(r'\{\s*([^,\}]+?)(\s*,\s*([^\}]+))?\s*\}')
imgref   = re.compile(r'\<\s*(\d+?)(\s*,\s*([^\>]*))?\s*\>')
```

For more on Python regular expressions have a look at `http://docs.python.org/py3k/library/re.html` or check the reading list in the appendix.

Next we define three utility functions, one for each pattern. Each function takes a `match` object that represents a matching pattern and returns a string that can be used in HTML to show or link to that reference:

Chapter6/wiki.py

```
def topicrefreplace(matchobj):
    ref=matchobj.group(1)
    txt=matchobj.group(3) if (not matchobj.group(3)
                              is None) else matchobj.group(1)
    nonexist = ""
    if(len(list(wikidb.Topic.list(title=ref)))==0):
           nonexist = " nonexisting"
```

```
    return '<a href="show?topic=%s" class="topicref%s">%s</a>'%(
                        ref,nonexist,txt)
def linkrefreplace(matchobj):
    ref=matchobj.group(1)
    txt=matchobj.group(3) if (not matchobj.group(3)
                              is None) else matchobj.group(1)
    ref=urlunparse(urlparse(ref,'http'))
    return '<a href="%s" class="externalref">%s</a>'%(ref,txt)
def imgrefreplace(matchobj):
    ref=matchobj.group(1)
    txt=matchobj.group(3) if (not matchobj.group(3)
                              is None) else matchobj.group(1)
    return '''<img src="showimage?id=%s" alt="%s"
                        class="wikiimage">'''%(ref,txt)
def render(content):
    yield '<p>\n'
    for line in content.splitlines(True):
          line = re.sub(imgref  ,imgrefreplace  ,line)
          line = re.sub(topicref,topicrefreplace,line)
          line = re.sub(linkref ,linkrefreplace ,line)
          if len(line.strip())==0 : line = '</p>\n<p>'
          yield line
    yield '</p>\n'
```

The `render()` function is passed a string with content to convert to HTML. For each line in the content, it tries to find the predefined patterns and converts them by passing the appropriate function to the `re.sub()` method. If a line consists of whitespace only, suitable HTML is produced to end a paragraph (highlighted).

Summary

We learned a lot in this chapter about implementing a web application that consists of more than a few entities and their relations.

Specifically, we covered:

- How to create a data model that describes entities and relations accurately
- How to create a delivery layer that is security conscious and treats incoming data with care
- How to use jQuery UI's dialog widget and extend the functionality of the autocomplete widget

We also encountered some limitations, especially in our entity/relation framework. It is, for example:

- Quite a lot of work to initialize the database as each entity and relation needs its own initialization code
- Unwieldy to specify things like sort order when retrieving entities
- Difficult to check input values or display formats in a uniform way
- Difficult to differentiate between different types of relations, like one-to-many or many-to-many

This hardly poses a problem for our moderately complex wiki application, but more complex applications can only be built with a more flexible framework—which is the topic of the next chapter.

7
Refactoring Code for Reuse

After doing a substantial bit of work, it is often a good idea to take a step back and look critically at your work to see if things could have been done better. Quite often, the insight gained in the development process can be used to good effect in any new code or even to refactor existing code if the benefits are so substantial that they warrant the extra work.

Often, such a critical reappraisal is motivated by observed shortcomings in the application, like poor performance, or by noting that requested changes take more time than we like because the code is designed in a less than optimal way.

Now that we have designed and implemented several applications in the last few chapters based on a simple entity/relation framework, it is time to have that critical look and see if there is room for improvement.

Time for action – taking a critical look

Examine each major piece of code (often a Python module that you implemented) and ask yourself the following questions:

- Could I reuse it without changes?
- How much extra code was needed to actually use the module?
- Did you really understand the documentation (even if you wrote it yourself)?
- How much of the code is duplicated?
- How easy is it to add new functionality?
- How well did it perform?

When we ask these questions about the entity and relation modules we developed, we see that:

- It was quite easy to reuse the modules
- But they do require quite a bit of extra code, for example, to initialize tables and threads

Also, we deliberately wrote specific modules to deal with domain-specific code, like input validation, but it is worth examining this code to see if we can discover patterns and enhance our framework to better support these patterns. One example is that we frequently require auto completion so it is worth looking at how this is implemented.

Performance-wise, we saw in the books application that we have hardly addressed the way in which large lists of books are browsed and this certainly needs attention if we wish to reuse our framework in settings that deal with large lists.

What just happened?

Now that we have pointed out several areas where our framework modules might be improved, it is time to consider if it is worth the effort.

Framework modules are intended to be reused by many applications, so a redesign that will allow the modules to be used with less additional code is a good idea, as less code means less maintenance. Of course, rewriting may mean that existing applications need to be rewritten if they want to use these new versions, but that is the price to pay for better maintenance.

Refactoring existing functionality is often less problematic, but benefits greatly from a good test framework to check if the new implementation still behaves as expected. Likewise, adding completely new functionality is likely to be even less of a problem, as existing applications do not yet use this functionality.

Because, in our case, we judge the advantages to outweigh the disadvantages, we will rework the entity/relation framework in the next section. We will not only focus on using less code, but also on making the definition of the new Entity and Relation classes easier to read. This will provide for a more intuitive use of these classes.

We will also devote a section to developing functionality to browse through lists of entities in a way that scales well, even if the lists are large and need to be sorted or filtered.

Refactoring

The first area that we will refactor is the way we can use the `Entity` class. Our goal is to enable a more intuitive use, without the need for explicit initialization of the database connections. To get a feeling for what is possible, let us first look at an example of how we would use the refactored `entity` module.

Time for action – defining new entities: how it should look

Type in and run the following code (also available as `testentity.py`). It will use the refactored `entity` module to define a `MyEntity` class and work with some instances of this class. We will create, list, and update instances and even see an update fail because we try to assign a value that will not pass a validation for an attribute:

Chapter7/testentity.py

```
from entity import *

class Entity(AbstractEntity):
    database="/tmp/abc.db"

class MyEntity(Entity):
    a=Attribute(unique=True, notnull=True, affinity='float',
          displayname='Attribute A', validate=lambda x:x<5)

a=MyEntity(a=3.14)
print(MyEntity.list())

e=MyEntity.list(pattern=[('a',3.14)])[0]
print(e)

e.delete()

a=MyEntity(a=2.71)
print([str(e) for e in MyEntity.list()])

a.a=1
a.update()
print([str(e) for e in MyEntity.list()])

a.a=9
```

The output produced by the print functions should look similar to the lines listed next, including the raised exception caused by an invalid update attempt:

```
[MyEntity(id=5)]
<MyEntity: Atrribute A=3.14, id=5>
['<MyEntity: Atrribute A=2.71, id=6>']
['<MyEntity: Atrribute A=1.0, id=6>']
Traceback (most recent call last):
```

```
  File "testentity.py", line 25, in <module>
    a.a=9
  File "C:\Documents and Settings\Michel\Bureaublad\MetaBase\Books II\
entity.py"
, line 117, in __setattr__
    raise AttributeError("assignment to "+name+" does not validate")
AttributeError: assignment to a does not validate
```

What just happened?

The first thing we note is that there is no explicit initialization of any database, nor is there any code to explicitly initialize any threads. All that is needed is to subclass the entity class from the `AbstractEntity` class provided in the entity module and define a database class variable that points to a file to be used as a database.

The next thing is that although we define a specific class (`MyEntity`, in this example) in a similar way as before by defining it as a subclass of Entity, we now specify any attributes by defining class variables that are assigned `Attribute` instances. In the example, we do this for just a single attribute `a` (highlighted). The `Attribute` instance encapsulates a lot of knowledge about constraints, and allows for the definition of a default value and a validation function.

Creating an instance isn't any different, but as the second `list()` example shows, this implementation allows for filtering so there is no need to retrieve all instance IDs, instantiate them as true objects and compare their attributes.

The final assignment to the `a` attribute shows the validation feature in action. It raises an `AttributeError` exception because trying to assign a value of 9 to it triggers our validation function.

These new and less cumbersome semantics are largely due to what can be achieved by using metaclasses, a concept we explore in the next section.

Metaclasses

Although the Python documentation warns that using metaclasses will make your head explode (read, for example, `http://www.python.org/doc/newstyle/`), they are not that dangerous: they may cause headaches, but these will mostly fade away after some experimenting and re-reading of the documentation.

Metaclasses allow you to inspect and alter the definition of a class just before this definition is made final and becomes available to the programmer. This is possible because, in Python, even classes are objects; specifically, they are instances of a metaclass. When we instantiate an object, we can control the way this instance is initialized by defining the `__init__()` and `__new__()` methods. Likewise, we can control the way a class is initialized by defining suitable `__init__()` and `__new__()` methods in its metaclass.

Just as all classes are ultimately subclasses of the `object` class, all metaclasses derive from the `type` metaclass. This means that if we want our class to be an instance of a different type of metaclass, we have to subclass `type` and define our class.

After reading the previous paragraphs, you may still fear your head may explode, but like most things, an example is much simpler to understand.

Time for action – using metaclasses

Say we want to be able to verify that classes we define always have an `__info__()` method. We can accomplish that by defining a suitable metaclass and defining any new class that should be checked with a reference to this metaclass. Look at the following example code (also available as `metaclassexample.py`):

Chapter7/metaclassexample.py

```
class hasinfo(type):
    def __new__(metaclass, classname, baseclasses, classdict):
        if len(baseclasses) and not '__info__' in classdict:
            raise TypeError('does not have __info__')
        return type.__new__(metaclass,
                 classname, baseclasses, classdict)
class withinfo(metaclass=hasinfo):
    pass
class correct(withinfo):
    def __info__(self): pass
class incorrect(withinfo):
    pass
```

This will raise an exception for the `incorrect` class, but not for the `correct` class.

What just happened?

As you can see, the `__new__()` method of a metaclass receives a number of important parameters. First, the metaclass itself and the `classname` of the class that is being defined, a (possibly empty) list of `baseclasses`, and finally the class dictionary. That last argument is very important.

As we define a class with a class statement, all methods and class variables we define here end up in a dictionary. Once this class is completely defined, this dictionary will be available as the `__dict__` attribute of the class. This is the dictionary that is passed to the `__new__()` method of the metaclass and we can examine and alter this dictionary at will.

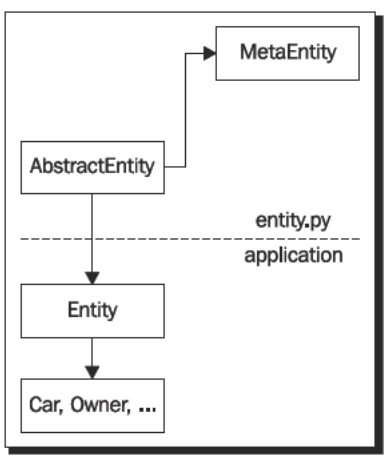
In this example, we simply check whether there exists a key called `__info__` in this class dictionary and raise an exception if it doesn't. (We do not really check that it is a method but this is possible too of course). If everything went well, we call the `__new__()` method of `type` (the mother of all metaclasses) as that method will take care of making the class definition available in the current scope.

There is an extra trick involved, however. The `withinfo` class is abstract in the sense that it defines the need for an `__info__()` method by referring to the `hasinfo` metaclass, but it does not define one itself. However, because it refers to the `hasinfo` metaclass, an exception would be raised because its own class dictionary is checked in the same way as its subclasses. To prevent this, we only check for the occurrence of the `__info__()` method if a class is a subclass, that is, when the list of base classes (available in the `baseclasses` parameter) is not empty.

Checking for mandatory methods is nice, but with so much information available, much more can be done. In the next section, we use this power to ensure that the definition of a new class will take care of creating suitable tables in a database backend as well.

MetaEntity and AbstractEntity classes

Besides creating a database table, if necessary, the metaclass that we will define will also examine the `Attribute` instance assigned to any class variable to build dictionaries of display names and validation functions. This way, subclasses can easily check if a column has such an attribute by using the column name as a key, thus obviating the need to check all class variables themselves again and again.



Time for action – implementing the MetaEntity and AbstractEntity classes

Let's see how this is done:

Chapter7/entity.py

```
class Attribute:
    def __init__(self, unique=False, notnull=False,
                               default=None, affinity=None, validate=None,
                               displayname=None, primary=False):
            self.coldef = (
                   (affinity+' ' if not affinity is None else '') +
                   ('unique '    if unique  else '') +
                   ('not null '  if notnull else '') +
                   ('default %s '%default if not default is None else '')
            )
            self.validate = validate
            self.displayname = displayname
            self.primary = primary
```

The `Attribute` class is mainly a vehicle to store information about attributes in a structured way. We could have used strings and parsed them, but by using an `Attribute` class, it is possible to explicitly recognize class variables that are meant to be attributes that are stored as database columns. That way, we can still define class variables that have a different purpose. Also, writing a parser is a lot of work, while checking parameters is a lot easier.

The highlighted code shows that most parameters are used to create a string that can be used as a column definition that is part of a create table statement. The other parameters (`displayname` and `validate`) are just stored as is for future reference:

Chapter7/entity.py

```
class MetaEntity(type):

  @classmethod
  def __prepare__(metaclass, classname, baseclasses, **kwds):
    return collections.OrderedDict()

  @staticmethod
  def findattr(classes,attribute):
    a=None
    for c in classes:
      if hasattr(c,attribute):
        a=getattr(c,attribute)
        break
```

```
    if a is None:
      for c in classes:
        a = MetaEntity.findattr(c.__bases__,attribute)
        if not a is None:
          break
      return a

  def __new__(metaclass,classname,baseclasses,classdict):
    def connect(cls):
      if not hasattr(cls._local,'conn'):
        cls._local.conn=sqlite.connect(cls._database)
        cls._local.conn.execute('pragma foreign_keys = 1')
        cls._local.conn.row_factory = sqlite.Row
      return cls._local.conn
    entitydefinition = False
    if len(baseclasses):
      if not 'database' in classdict:
        classdict['_database']=MetaEntity.findattr(
                    baseclasses,'database')
        if classdict['_database'] is None:
          raise AttributeError(
            '''subclass of AbstractEntity has no
            database class variable''')
        entitydefinition=True
      if not '_local' in classdict:
        classdict['_local']=MetaEntity.findattr(
                    baseclasses,'_local')
      classdict['_connect']=classmethod(connect)
      classdict['columns']=[
        k for k,v in classdict.items()
          if type(v) == Attribute]
      classdict['sortorder']=[]
      classdict['displaynames']={
        k:v.displayname if v.displayname else k
        for k,v in classdict.items()
        if type(v) == Attribute}
      classdict['validators']={
        k:v.validate for k,v in classdict.items()
        if type(v) == Attribute
          and not v.validate is None}
      classdict['displaynames']['id']='id'
      PrimaryKey = Attribute()
```

```
        PrimaryKey.coldef = 'integer primary key '
        PrimaryKey.coldef+= 'autoincrement'
        if entitydefinition:
          sql = 'create table if not exists '
          sql+= classname +' ('
          sql+= ", ".join([k+' '+v.coldef
            for k,v in [('id',PrimaryKey)]
                  +list(classdict.items())
                  if type(v) == Attribute])
          sql+= ')'
          conn = sqlite.connect(classdict['_database'])
          conn.execute(sql)
        for k,v in classdict.items():
          if type(v) == Attribute:
            if v.primary:
              classdict['primary']=property(
                lambda self:getattr(self,k))
              classdict['primaryname']=k
              break
        if not 'primary' in classdict:
          classdict['primary']=property(
            lambda self:getattr(self,'id'))
          classdict['primaryname']='id'
      return type.__new__(metaclass,
        classname,baseclasses,classdict)
```

The metaclass we will use to synchronize the creation of database tables and the creation of entity classes is called `MetaEntity`. Its `__new__()` method is where all the action takes place, but there is one important additional method: `__prepare__()`.

The `__prepare__()` method is called to provide an object that can be used as a class dictionary. The default, as provided by the `type` class, just returns a regular Python `dict` object. Here we return an ordered dictionary, a dictionary that will remember the order of its keys as they are entered. This will enable us to use the order in which class variables are declared, for example, to use this as the default order to display columns. Without an ordered dictionary, we wouldn't have any control and would have to supply separate information.

The `__new__()` method first checks if we are a subclass of `MetaEntity` by checking whether the list of base classes is non zero (highlighted) as `MetaEntity` itself does not have a database backend.

Then it checks if the database class variable is defined. If not, we are a specific entity that has a database backend and we try to locate the database class variable in one of our super classes. If we find it, we store it locally; if not, we raise an exception because we cannot function without a reference to a database.

The `AbstractEntity` class will have a `_local` class variable defined that holds a reference to thread local storage, and subclasses will have their own `_local` variable that points to the same thread local storage.

The next step is to gather all sorts of information from all the class variables that refer to `Attribute` instances. First we collect a list of column names (highlighted). Remember that because we caused the class dictionary to be an ordered dictionary, these column names will be in the order they were defined.

Likewise, we define a list of display names. If any attribute does not have a `displayname` attribute, its display name will be identical to its column name. We also construct a dictionary of validators, that is, a dictionary indexed by column name that holds a function to validate any value before it is assigned to a column.

Every entity will have an `id` attribute (and a corresponding column in the database table) that is created automatically without it being explicitly defined. Therefore, we add its `displayname` separately and construct a special Attribute instance (highlighted).

This `coldef` attribute of this special `Attribute` together with the `coldef` attributes of the other `Attribute` instances will then be used to compose an SQL statement that will create a table with the proper column definitions.

Finally, we pass the altered and augmented class dictionary together with the original list of base classes and the class name to the `__new__()` method of the type class which will take care of the actual construction of the class.

The rest of the functionality of any `Entity` is not implemented by its metaclass, but in the regular way, that is, by providing methods in the class that all entities should derive from: `AbstractEntity`:

Chapter7/entity.py

```
class AbstractEntity(metaclass=MetaEntity):
    _local = threading.local()
    @classmethod
    def listids(cls,pattern=None,sortorder=None):
           sql = 'select id from %s'%(cls.__name__,)
           args = []
           if not pattern is None and len(pattern)>0:
                  for s in pattern:
                         if not (s[0] in cls.columns or s[0]=='id'):
                                raise TypeError('unknown column '+s[0])
                  sql += " where "
                  sql += " and ".join("%s like ?"%s[0] for s in
pattern)
```

```
                args+= [s[1] for s in pattern]
        if sortorder is None:
                if not cls.sortorder is None :
                        sortorder = cls.sortorder
        else:
                for s in sortorder:
                        if not (s[0] in cls.columns or s[0]=='id'):
                                raise TypeError('unknown column '+s[0])
                        if not s[1] in ('asc', 'desc') :
                                raise TypeError('illegal sort
argument'+s[1])
        if not (sortorder is None or len(sortorder) == 0):
                sql += ' order by '
                sql += ','.join(s[0]+' '+s[1] for s in sortorder)
        cursor=cls._connect().cursor()
        cursor.execute(sql,args)
        return [r['id'] for r in cursor]

    @classmethod
    def list(cls,pattern=None,sortorder=None):
        return [cls(id=id) for id in cls.listids(
                sortorder=sortorder,pattern=pattern)]

    @classmethod
    def getcolumnvalues(cls,column):
        if not column in cls.columns :
                raise KeyError('unknown column '+column)
        sql ="select %s from %s order by lower(%s)"
        sql%=(column,cls.__name__,column)
        cursor=cls._connect().cursor()
        cursor.execute(sql)
        return [r[0] for r in cursor.fetchall()]

    def __str__(self):
        return '<'+self.__class__.__name__+': '+", ".join(
                ["%s=%s"%(displayname, getattr(self,column))
                 for column,displayname
                 in self.displaynames.items()])+'>'

    def __repr__(self):
        return self.__class__.__name__+"(id="+str(self.id)+")"

    def __setattr__(self,name,value):
        if name in self.validators :
                if not self.validators[name](value):
                        raise AttributeError(
                                "assignment to "+name+" does not
```

```
validate")
            object.__setattr__(self,name,value)
    def __init__(self,**kw):
            if 'id' in kw:
                    if len(kw)>1 :
                            raise AttributeError('extra keywords besides
id')
                    sql = 'select * from %s where id = ?'
                    sql%= self.__class__.__name__
                    cursor = self._connect().cursor()
                    cursor.execute(sql,(kw['id'],))
                    r=cursor.fetchone()
                    for c in self.columns:
                            setattr(self,c,r[c])
                    self.id = kw['id']
            else:
                    rels={}
                    attr={}
                    for col in kw:
                            if not col in self.columns:
                                    rels[col]=kw[col]
                            else:
                                    attr[col]=kw[col]
                    name = self.__class__.__name__
                    cols = ",".join(attr.keys())
                    qmarks = ",".join(['?']*len(attr))
                    if len(cols):
                            sql = 'insert into %s (%s) values (%s)'
                            sql%= (name,cols,qmarks)
                    else:
                            sql = 'insert into %s default values'%name
                    with self._connect() as conn:
                            cursor = conn.cursor()
                            cursor.execute(sql,tuple(attr.values()))
                            self.id = cursor.lastrowid
    def delete(self):
            sql = 'delete from %s where id = ?'
            sql%= self.__class__.__name__
            with self._connect() as conn:
                            cursor = conn.cursor()
                            cursor.execute(sql,(self.id,))
    def update(self,**kw):
            for k,v in kw.items():
```

```
            setattr(self,k,v)
    sets = []
    vals = []
    for c in self.columns:
            if not c == 'id':
                    sets.append(c+'=?')
                    vals.append(getattr(self,c))
    table = self.__class__.__name__
    sql = 'update %s set %s where id = ?'
    sql%=(table,",".join(sets))
    vals.append(self.id)
    with self._connect() as conn:
                    cursor = conn.cursor()
                    cursor.execute(sql,vals)
```

What just happened

`AbstractEntity` provides a number of methods to provide CRUD functionality:

- A constructor to refer to the existing entities in the database or to create new ones
- `list()` and `listids()`, to find instances that match certain criteria
- `update()`, to synchronize changed attributes of an entity with the database
- `delete()`, to delete an entity from the database

It also defines the 'special' Python methods `__str__()`, `__repr__()`, and `__setattr__()` to render an entity in a legible way and to validate the assignment of a value to an attribute.

Obviously, `AbstractEntity` refers to the `MetaEntity` metaclass (highlighted). It also defines a `_local` class variable that refers to thread local storage. The `MetaEntity` class will make sure this reference (but not its contents) are copied to all subclasses for fast access. By defining it here, we will make sure that all subclasses refer to the same thread local storage, and more importantly, will use the same connection to the database for each thread instead of using a separate database connection for each different entity.

The `listids()` class method will return a list of IDs of entities that match the criteria in its `pattern` argument or the IDs of all the entities if no criteria were given. It will use the `sortorder` argument to return the list of IDs in the required order. Both `sortorder` and `pattern` are a list of tuples, each tuple having the column name as its first item. The second item will be a string to match against for the `pattern` argument or either `asc` or `desc` for the `sortorder` argument, signifying an ascending or descending sort respectively.

The SQL statement to retrieve any matching IDs is constructed by first creating the `select` part (highlighted), as this will be the same irrespective of any additional restrictions. Next, we check if there are any `pattern` components specified, and if so, add a `where` clause with matching parts for each `pattern` item. The matches we specify use the SQL `like` operator, which is normally only defined for strings, but SQLite will convert any argument to a string if we use the like operator. Using the like operator will allow us to use SQL wildcards (for example, `%`).

The next stage is to check if there were any sort order items specified in the `sortorder` argument. If not, we use the default sort order stored in the `sortorder` class variable (which in our current implementation will still be `None`). If there are items specified, we add an order by clause and add specifications for each sort item. A typical SQL statement will look something like `select id from atable where col1 like ? and col2 like ? order by col1 asc`.

Finally, we use the `_connect()` method (that was added by the metaclass) to retrieve a database connection (and establish one, if needed) that we can use to execute the SQL query with and retrieve the list of IDs.

The `list()` method bears a close resemblance to the `listids()` method and takes the same arguments. It will, however, return a list of entity instances rather than a list of just the IDs by calling the entity's constructor with each ID as an argument. This is convenient if that was what we wanted to do with those IDs anyway, but a list of IDs is often easier to manipulate. We therefore provide both methods.

Being able to retrieve lists of entities is nice, but we must also have a means to create a new instance and to retrieve all information associated with a record with a known ID. That is where the entity's constructor, in the form of the `__init__()` method, comes in.

Strictly speaking, `__init__()` isn't a constructor but a method that initializes an instance after it is constructed.

If a single ID keyword argument is passed to `__init__()` (highlighted), all columns of the record matching that ID are retrieved and the corresponding arguments (that is, attributes with the same name as a column) are set using the `setattr()` built-in function.

If more than one keyword argument is passed to `__init__()`, each should be the name of a defined column. If not, an exception is raised. Otherwise, the keywords and their values are used to construct an insert statement. If there are more columns defined than there are keywords given, this will result in default values to be inserted. The default is usually `NULL` unless a `default` argument was specified for the column when the `Entity` was defined. If `NULL` is the default and the column has a `non null` constraint, an exception will be raised.

Because ID columns are defined as `autoincrement` columns and we do not specify an explicit ID value, the ID value will be equal to the `rowid`, a value we retrieve as the `lastrowid` attribute of the cursor object (highlighted).

Have a go hero – retrieving instances in a single step

Retrieving a list of IDs first and then instantiating entities means that we have to retrieve all attributes of each entity with a separate SQL statement. This might have a negative impact on performance if the list of entities is large.

Create a variant of the `list()` method that will not convert the selected IDs to entity instances one-by–one, but will use a single select statement to retrieve all attributes and use those to instantiate entities.

Relations

Defining relations between entities should be just as hassle free as defining the entities themselves, and with the power of metaclasses in our hands, we can use the same concepts. But let's first have a look at how we would use such an implementation.

Time for action – defining new relations: how it should look

The following sample code shows how we would use a `Relation` class (also available in `relation.py`):

Chapter7/relation.py

```
from os import unlink
db="/tmp/abcr.db"
try:
      unlink(db)
except:
      pass
class Entity(AbstractEntity):
      database=db
class Relation(AbstractRelation):
      database=db
class A(Entity): pass
class B(Entity): pass
class AB(Relation):
      a=A
      b=B
```

```
a1=A()
a2=A()
b1=B()
b2=B()

a1.add(b1)
a1.add(b2)
print(a1.get(B))
print(b1.get(A))
```

What just happened?

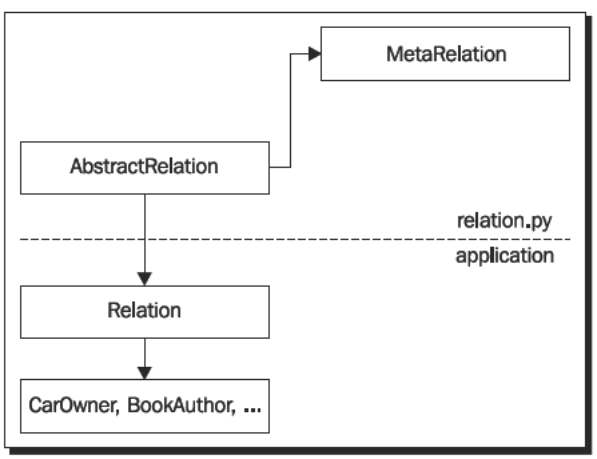
After defining a few entities, defining the relation between those entities follows the same pattern: we define a `Relation` class that is a subclass of `AbstractRelation` to establish a reference to a database that will be used.

Then we define an actual relation between two entities by subclassing `Relation` and defining two class variables, `a` and `b` that refer to the `Entity` classes that form each half of the relation.

If we instantiate a few entities, we may then define a few relations between these instances by using the `add()` method and retrieve related entities with the `get()` method.

Note that those methods are called on the `Entity` instances, allowing for a much more natural idiom than the use of class methods in the `Relation` class. These `add()` and `get()` methods were added to those entity classes by the `MetaRelation` metaclass, and in the next section, we will see how this is accomplished.

The class diagram for relation classes looks nearly identical to the one for entities:



Implementing the MetaRelation and AbstractRelation classes

The implementation of the `AbstractRelation` class is very minimalistic because it is only used to create some thread local storage and establish a relation with the `MetaRelation` metaclass:

Chapter7/relation.py

```
class AbstractRelation(metaclass=MetaRelation):
    _local = threading.local()
```

No methods are specified since the metaclass will take care of adding suitable methods to the entity class that are a part of this relationship.

The `MetaRelation` class has two goals: creating a database table that will hold records for each individual relation and adding methods to the entity classes involved, so that relations can be created, removed, and queried:

Chapter7/relation.py

```
class MetaRelation(type):
  @staticmethod
  def findattr(classes,attribute):
    a=None
    for c in classes:
      if hasattr(c,attribute):
        a=getattr(c,attribute)
        break
    if a is None:
      for c in classes:
        a = MetaRelation.findattr(c.__bases__,attribute)
        if not a is None:
          break
    return a

  def __new__(metaclass,classname,baseclasses,classdict):
    def connect(cls):
      if not hasattr(cls._local,'conn'):
        cls._local.conn=sqlite.connect(cls._database)
        cls._local.conn.execute('pragma foreign_keys = 1')
        cls._local.conn.row_factory = sqlite.Row
      return cls._local.conn
    def get(self,cls):
      return getattr(self,'get'+cls.__name__)()
    def getclass(self,cls,relname):
```

```
    clsname = cls.__name__
    sql = 'select %s_id from %s where %s_id = ?'%(
      clsname,relname,self.__class__.__name__)
    cursor=self._connect().cursor()
    cursor.execute(sql,(self.id,))
    return [cls(id=r[clsname+'_id']) for r in cursor]

  def add(self,entity):
    return getattr(self,
      'add'+entity.__class__.__name__)(entity)

  def addclass(self,entity,Entity,relname):
    if not entity.__class__ == Entity :
      raise TypeError(
        'entity not of the required class')

    sql = 'insert or replace into %(rel)s '
    sql+= '(%(a)s_id,%(b)s_id) values (?,?)'
    sql%= { 'rel':relname,
        'a':self.__class__.__name__,
        'b':entity.__class__.__name__}
    with self._connect() as conn:
      cursor = conn.cursor()
      cursor.execute(sql,(self.id,entity.id))

  relationdefinition = False
  if len(baseclasses):
    if not 'database' in classdict:
      classdict['_database']=MetaRelation.findattr(
        baseclasses,'database')
      if classdict['_database'] is None:
        raise AttributeError(
          '''subclass of AbstractRelation has no
          database class variable''')
      relationdefinition=True

    if not '_local' in classdict:
      classdict['_local']=MetaRelation.findattr(
        baseclasses,'_local')

    classdict['_connect']=classmethod(connect)

    if relationdefinition:
      a = classdict['a']
      b = classdict['b']
      if not issubclass(a,AbstractEntity) :
        raise TypeError('a not an AbstractEntity')
      if not issubclass(a,AbstractEntity) :
        raise TypeError('b not an AbstractEntity')
```

```
            sql = 'create table if not exists %(rel)s '
            sql+= '( %(a)s_id references %(a)s '
            sql+= 'on delete cascade, '
            sql+= '%(b)s_id references %(b)s '
            sql+= 'on delete cascade, '
            sql+= 'unique(%(a)s_id,%(b)s_id))'
            sql%= { 'rel':classname,
                'a':a.__name__,
                'b':b.__name__}
            conn = sqlite.connect(classdict['_database'])
            conn.execute(sql)
            setattr(a,'get'+b.__name__,
              lambda self:getclass(self,b,classname))
            setattr(a,'get',get)
            setattr(b,'get'+a.__name__,
              lambda self:getclass(self,a,classname))
            setattr(b,'get',get)
            setattr(a,'add'+b.__name__,
            lambda self,entity:addclass(self,
                      entity,b,classname))
            setattr(a,'add',add)
            setattr(b,'add'+a.__name__,
            lambda self,entity:addclass(self,
                        entity,a,classname))
            setattr(b,'add',add)
        return type.__new__(metaclass,
                    classname,baseclasses,classdict)
```

As was the case for the `MetaEntity` class, `MetaRelation` performs its magic through its `__new__()` method.

First, we check if we are creating a subclass of `AbstractRelation` by checking the length of the `baseclasses` parameter (remember that `MetaRelation` is defined as a metaclass for `AbstractRelation`, meaning that not only its subclasses, but also `AbstractRelation` itself will be processed by the metaclass machinery, something that is not really needed here).

If it is a subclass, we copy the database and thread local storage references to the class dictionary for quick access.

If there was no `database` attribute specified, we know the class being defined is a subclass of `Relation`, that is, a specific relation class and mark this in the `relationdefinition` variable (highlighted).

If we are dealing with a concrete definition of a relation, we will have to work out which entities are involved. This is done by checking the class dictionary for attributes named `a` and `b`, that should be subclasses of `AbstractEntity` (highlighted). These are both halves of the relation and their names are used to create a bridging table if not already present.

If we were to define a relation like this:

```
class Owner(Relation):
    a=Car
    b=User
```

The SQL statement generated would be:

```
create table if not exists Owner (
    Car_id references Car on delete cascade,
    User_id references User on delete cascade,
    unique(Car_id,User_id)
)
```

Each column references the primary key in the corresponding table (because we did specify just the table in the references clause) and the `on delete cascade` constraint will make sure that if an entity is deleted, the relation is deleted as well. The final `unique` constraint will make sure that if there is a relation between specific instances, there will be only one record reflecting this.

Adding new methods to existing classes

The final part of the `__new__()` method deals with inserting methods in the entity classes that are involved in this relation. Adding methods to other classes may sound like magic, but in Python, classes themselves are objects too and have class dictionaries that hold the attributes of a class. Methods are just attributes that happen to have a value that is a function definition.

We can, therefore, add a new method at runtime to any class by assigning a reference to a suitable function to a class attribute. The `MetaEntity` class only altered the class dictionary of an `Entity` before it was created. The `MetaRelation` class goes one step further and not only alters the class dictionary of the `Relation` class, but also those of the `Entity` classes involved.

Altering class definitions at runtime is not limited to metaclasses, but should be used sparingly because we expect classes to behave consistently anywhere in the code.

If we have two classes, `A` and `B`, we want to make sure each has its own complement of get and add methods. That is, we want to make sure the `A` class has `getB()` and `addB()` methods and the `B` class has `getA()` and `addA()`. We, therefore, define generic `getclass()` and `addclass()` functions and assign those with tailored lambda functions to the named attributes in the class concerned (highlighted).

If we assume again that the entity classes are called `A` and `B` and our relation is called `AB`, the assignment:

```
setattr(a,'get'+b.__name__,lambda self:getclass(self,b,classname))
```

will mean that the A class will now have a method called `getB` and if that method is called on an instance of `A` (like `a1.getB()`) it will result in a call to `getclass` like:

```
getclass(a1,B,'AB')
```

We also create (or redefine) a `get()` method that when given a class as an argument will find the corresponding `getXXX` method.

The `getclass()` method is defined as follows:

```
            def getclass(self,cls,relname):
                  clsname = cls.__name__
                  sql = 'select %s_id from %s where %s_id =
?'%(clsname,relname,self.__class__.__name__)
                  cursor=self._connect().cursor()
                  cursor.execute(sql,(self.id,))
                  return [cls(id=r[clsname+'_id']) for r in cursor]
```

First, it constructs an SQL statement. If `getclass()` was invoked like `getclass(a1,B,'AB')`, this statement might look like this:

```
select B_id from AB where A_id = ?
```

Then it executes this statement with `self.id` as the argument. The resulting list of IDs is returned as a list of instances.

The add functionality follows the same pattern, so we only take a quick look at the `addclass()` function. It first checks if the entity we are trying to add is of the required class. Note that if we make a call like `a1.addB(b1)`, it will refer to a function inserted by the `MetaRelation` class that will then be called like `addclass(a1,b1,B,'AB')`.

The SQL statement that is subsequently constructed may look like this:

```
insert or replace into AB (A_id,B_id) values (?,?)
```

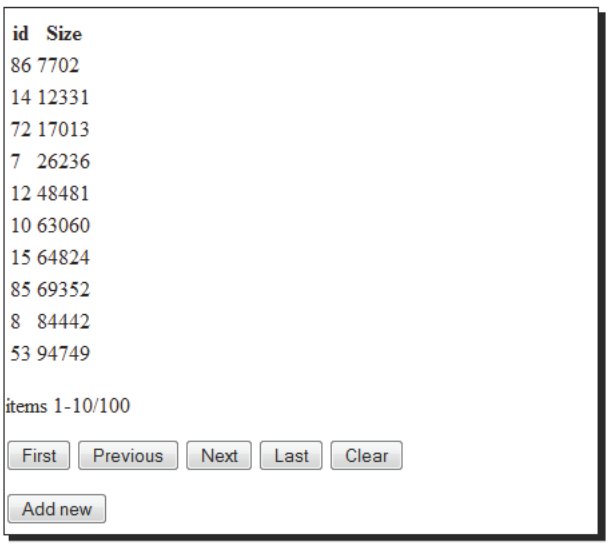
Because of the unique constraint we specified earlier, a second insert that specifies the same specific relation may fail in which case we replace the record (that is effectively ignoring the failure). This way, we may call `add()` twice with the same arguments, yet still end up with just a single record of the relation.

Browsing lists of entities

One of the most important tools for a user to interact with a collection of entities is a table. A table provides a logical interface to page through lists of data and present relevant attributes in columns. Other features often found in such a table interface are the options to sort on one or more attributes and to drill down, that is, to show only those entities that have some specific value for an attribute.

Time for action – using a table-based Entity browser

Run `browse.py` and point your browser to `http://localhost:8080`. A small sample application is started that shows lists of random data, as can be seen in the following image:



This rather Spartan looking interface may lack most visual adornments, but it is fully functional nevertheless. You may page through the list of data by clicking the appropriate buttons in the button bar at the bottom, change the sort order of the list by clicking one or more times on a header (which will cycle through ascending, descending, or no sort at all, however, without any visual feedback at the moment) or reduce the list of items shown by clicking on a value in a column, that will result in a list of items that share the same value in this column. All items may be shown again by clicking the **Clear** button.

What just happened?

The `browse` module (which is available as `browse.py`) contains more than the sample application. It also defines a reusable `Browse` class that can be initialized with a reference to an `Entity` and used as a CherryPy application. The `Browse` class can also be given arguments that specify which, if any, columns should be shown.

Its intended use is best illustrated by taking a look at the sample application:

Chapter7/browse.py

```
from random import randint
import os

current_dir = os.path.dirname(os.path.abspath(__file__))

class Entity(AbstractEntity):
        database='/tmp/browsetest.db'

class Number(Entity):
        n = Attribute(displayname="Size")

n=len(Number.listids())
if n<100:
        for i in range(100-n):
                Number(n=randint(0,1000000))

root = Browse(Number, columns=['id','n'],
        sortorder=[('n','asc'),('id','desc')])

cherrypy.quickstart(root,config={
        '/':
        { 'log.access_file' :
                os.path.join(current_dir,"access.log"),
        'log.screen': False,
        'tools.sessions.on': True
        }
})
```

It initializes an instance of the `Browse` class with a single mandatory argument as a subclass of `Entity`, in this case, `Number`. It also takes a `columns` argument that takes a list that specifies which attributes to show in the table's columns and in which order. It also takes a `sortorder` argument, a list of tuples that specifies on which columns to sort and in which direction.

This instance of the `Browse` class is then passed to CherryPy's `quickstart()` function to deliver the functionality to the client. It would be just as simple to mount two different `Browse` instances, each servicing a different `Entity` class within a custom root application.

How is all this implemented? Let's first take a look at the `__init__()` method:

Chapter7/browse.py

```
class Browse:
   def __init__(self,entity,columns=None,
          sortorder=None,pattern=None,page=10,show="show"):
          if not issubclass(entity,AbstractEntity) :
                raise TypeError()
          self.entity = entity
          self.columns = entity.columns if columns is None else
columns
          self.sortorder = [] if sortorder is None else sortorder
          self.pattern = [] if pattern is None else pattern
          self.page = page
          self.show = show
          self.cache= {}
          self.cachelock=threading.Lock()
          self.cachesize=3
          for c in self.columns:
                if not (c in entity.columns or c == 'id') and not (
                            hasattr(self.entity,'get'+c.__name__)) :
                      raise ValueError('column %s not defined'%c)
          if len(self.sortorder) > len(self.columns) :
                raise ValueError()
          for s in self.sortorder:
                if s[0] not in self.columns and s[0]!='id':
                      raise ValueError(
                            'sorting on column %s not
possible'%s[0])
                if s[1] not in ('asc','desc'):
                      raise ValueError(
                            'column %s, %s is not a valid sort
order'%s)
          for s in self.pattern:
                if s[0] not in self.columns and s[0]!='id':
                      raise ValueError(
                            'filtering on column %s not
possible'%s[0])
          if self.page < 5 :
                      raise ValueError()
```

The `__init__()` method takes quite a number of arguments and only the `entity` argument is mandatory. It should be a subclass of `AbstractEntity` and this is checked in the highlighted code.

All parameters are stored and initialized to suitable defaults if missing.

The `columns` argument defaults to a list of all columns defined for the entity, and we verify that any column we want to display is actually defined for the entity.

Likewise, we verify that the `sortorder` argument (a list of tuples containing the column name and its sort direction) contains no more items than there are columns (as it is not sensible to sort more than once on the same column) and that the sort directions specified are either `asc` or `desc` (for ascending and descending respectively).

The `pattern` argument, a list of tuples containing the column name and a value to filter on, is treated in a similar manner to see if only defined columns are filtered on. Note that it is perfectly valid to filter or sort on a column or columns that are themselves not shown. This way, we can display subsets of a large dataset without bothering with too many columns.

The final sanity check is done on the `page` argument which specifies the number of rows to show on each page. Very few rows feels awkward and negative values are meaningless, so we settle for a lower limit of five rows per page:

Chapter7/browse.py

```
@cherrypy.expose
def index(self, _=None, start=0,
  pattern=None, sortorder=None, cacheid=None,
  next=None,previous=None, first=None, last=None,
  clear=None):
  if not clear is None :
    pattern=None
  if sortorder is None :
    sortorder = self.sortorder
  elif type(sortorder)==str:
    sortorder=[tuple(sortorder.split(','))]
  elif type(sortorder)==list:
    sortorder=[tuple(s.split(',')) for s in sortorder]
  else:
    sortorder=None
  if pattern is None :
    pattern = self.pattern
  elif type(pattern)==str:
    pattern=[tuple(pattern.split(','))]
  elif type(pattern)==list:
    pattern=[tuple(s.split(',',1)) for s in pattern]
  else:
    pattern=None
  ids = self.entity.listids(
```

```
            pattern=pattern,sortorder=sortorder)
      start=int(start)
      if not next is None :
        start+=self.page
      elif not previous is None :
        start-=self.page
      elif not first is None :
        start=0
      elif not last is None :
        start=len(ids)-self.page
      if start >= len(ids) :
        start=len(ids)-1
      if start<0 :
        start=0
      yield '<table class="entitylist" start="%d" page="%d">\
n'%(start,self.page)
      yield '<thead><tr>'
      for col in self.columns:
        if type(col) == str :
          sortclass="notsorted"
          for s in sortorder:
            if s[0]==col :
              sortclass='sorted-'+s[1]
              break
          yield '<th class="%s">'%sortclass+self.entity.
displaynames[col]+'</th>'
        else :
          yield '<th>'+col.__name__+'</th>'
      yield '</tr></thead>\n<tbody>\n'
      entities = [self.entity(id=i)
              for i in ids[start:start+self.page]]
      for e in entities:
        vals=[]
        for col in self.columns:
          if not type(col) == str:
            vals.append(
              "".join(
              ['<span class="related" entity="%s" >%s</span> ' % (r.__
class__.__name__, r.primary) for r in e.get(col)]))
          else:
            vals.append(str(getattr(e,col)))
        yield ('<tr id="%d"><td>'
            + '</td><td>'.join(vals)+'</td></tr>\n')%(e.id,)
      yield '</tbody>\n</table>\n'
```

```
        yield '<form method="GET" action=".">'
        yield '<div class="buttonbar">'
        yield '<input name="start" type="hidden" value="%d">\n'%start
        for s in sortorder:
          yield '<input name="sortorder" type="hidden" value="%s,%s">\n'%s
        for f in pattern:
          yield '<input name="pattern" type="hidden" value="%s,%s">\n'%f
        yield '<input name="cacheid" type="hidden" value="%s">'%cacheid
        yield '<p class="info">items %d-%d/%d</p>'%(start+1,start+len
    (entities),len(ids))
        yield '<button name="first" type="submit">First</button>\n'
        yield '<button name="previous" type="submit">Previous</button>\n'
        yield '<button name="next" type="submit">Next</button>\n'
        yield '<button name="last" type="submit">Last</button>\n'
        yield '<button name="clear" type="submit">Clear</button>\n'
        yield '</div>'
        yield '</form>'
        # no name attr on the following button otherwise it may be sent as
    an argument!
        yield '<form method="GET" action="add"><button type="submit">Add
    new</button></form>'
```

Both the initial display of the table as well as paging, sorting, and filtering are taken care of by the same `index()` method. To understand all the parameters it may take, it might be helpful to look at the HTML markup it produces for our sample application.

The `index()` method of the `Browse` class is not the only place where we encounter a fair amount of HTML to be delivered to the client. This might become difficult to read and therefore difficult to maintain and using templates might be a better solution. A good start point for choosing a template solution that works well with CherryPy is `http://www.cherrypy.org/wiki/ChoosingATemplatingLanguage`.

Time for action – examining the HTML markup

Let's have a look at how the HTML markup produced by the `index()` method looks:

```
<table class="entitylist" start="0" page="10">
   <thead>
         <tr>
               <th class="sorted-desc">id</th>
               <th class="sorted-asc">Size</th>
         </tr>
   </thead>
```

```
    <tbody>
        <tr id="86"><td>86</td><td>7702</td></tr>
        <tr id="14"><td>14</td><td>12331</td></tr>
        <tr id="72"><td>72</td><td>17013</td></tr>
        <tr id="7"><td>7</td><td>26236</td></tr>
        <tr id="12"><td>12</td><td>48481</td></tr>
        <tr id="10"><td>10</td><td>63060</td></tr>
        <tr id="15"><td>15</td><td>64824</td></tr>
        <tr id="85"><td>85</td><td>69352</td></tr>
        <tr id="8"><td>8</td><td>84442</td></tr>
        <tr id="53"><td>53</td><td>94749</td></tr>
    </tbody>
</table>
<form method="GET" action=".">
    <div class="buttonbar">
        <input name="start" type="hidden" value="0">
        <input name="sortorder" type="hidden" value="n,asc">
        <input name="sortorder" type="hidden" value="id,desc">
        <input name="cacheid" type="hidden"
               value="57ec8e0a53e34d428b67dbe0c7df6909">
        <p class="info">items 1-10/100</p>
        <button name="first" type="submit">First</button>
        <button name="previous" type="submit">Previous</button>
        <button name="next" type="submit">Next</button>
        <button name="last" type="submit">Last</button>
        <button name="clear" type="submit">Clear</button>
    </div>
</form>
<form method="GET" action="add">
    <button type="submit">Add new</button>
</form>
```

Apart from the actual table, we have a `<form>` element with quite a number of `<button>` and `<input>` elements, albeit that most have their type attribute set to hidden.

The `<form>` element has an action attribute " . " (a single dot), which will cause all the information in the form to be submitted to the same URL that originated this form, so the data will be processed by the same `index()` method we are now examining. A submit is triggered when any of the `<button>` elements with a `type` attribute equal to `submit` is clicked, in which case, not only the `<input>` elements are sent, but also the name of the button that was clicked.

Note that any `<input>` element that has to be sent to the server should have a `name` attribute. Omitting the `name` attribute will cause it to be missed out. `<input>` elements with `type` equal to `hidden` are sent as well if they have a `name` attribute. Hidden `<input>` elements are not displayed, but do play an important role in keeping essential information associated with a form together.

The first hidden `<input>` element in the form stores the start index of the items currently displayed in the table. By adding it as a hidden element, we can calculate which items to show when we take action when the **Next** or **Previous** button is clicked.

We also want to remember if and how the items are sorted. Therefore, we include a number of hidden input elements with a `name` attribute equal to `sortorder`, each having a value consisting of a column name and a sort direction separated by a comma.

When a form is submitted, input elements with the same name are added in order as arguments to the `action` URL and CherryPy will recognize this pattern and convert them to a list of values. In this example, the `index()` method of the `Browse` class receives this list as its `sortorder` argument. Any `pattern` values are present as hidden `<input>` elements as well and processed in an identical way.

The form also contains an `info` class `<p>` element, that contains information on the number of items and the items actually shown on the current page. The final part of the form is a collection of submit buttons.

What just happened?

The `index()` method may be called with no arguments at all or with any or all contents of the form it displays. If the client-side JavaScript code wants to call it asynchronously while preventing the browser from caching it, it may even pass an _ (underscore) argument with a random value, which will be ignored.

The rest of the arguments are relevant and checked for sanity before being acted upon.

We want the `sortorder` variable to contain a list of tuples, each consisting of a column name and a sort direction, but the values of the input elements are simply interpreted as strings by CherryPy, so we have to convert this list of strings to a list of tuples by splitting those strings on the comma separator. We neither check for the validity of the column names, nor for that of the sort directions because that will be done by the code doing the actual work.

The `pattern` variable is treated in a similar way, but because we may want to filter on values containing commas, we cannot simply use the `split()` method here, but have to pass it a limit of 1 to restrict its splitting to the first comma it encounters.

Next, we pass the `sortorder` and `pattern` variables to the `listids()` class method of the entity we stored with this `Browse` instance. It will return a list of IDs of instances that match the `pattern` criteria (or all instances if no patterns are specified) sorted in the correct order. Note that since the number of instances might be huge, we do not use the `list()` method here because converting all IDs to entity instances at once might render the application unresponsive. We will just convert those IDs to instances that we will actually show on the page, based on the start and page variables.

To calculate the new start index, we will have to check if we act upon one of the paging buttons (highlighted) and add or subtract a page length if we are acting on a click on the **Next** or **Previous** button. We set the start index to 0 if the **First** button was clicked. If the **Last** button was clicked, we set the start index to the number of items minus the length of the page. If any of these calculations result in a start index that is less than zero, we set it to zero.

The next step is to produce the actual output, yielding one line at a time, beginning with the `<table>` element. Our table consists of a head and a body, the head consisting of a single row of `<th>` elements, each containing either the display name of the column we are showing if it represents an attribute of the entity, or the name of the class if it represents a related entity. Any sort order associated with this column is represented in its `class` attribute, so we may use CSS to make this visible to the user.

To display the rows in the body of the table, we convert the relevant IDs in the selection to actual entities (highlighted) and generate `<td>` elements for each attribute. If the column refers to related entities, their primary attributes are displayed, each related entity encapsulated in its own `<span>` element. The latter will enable us to associate relevant actions with each individual item shown, for example, displaying it in full when it is clicked.

The final long list of `yield` statements is used to produce the form with its many hidden input elements, each recording the arguments that were passed to the `index()` method.

Caching

The bulk of the activity when browsing through lists in a typical application is paging forward and backward. If we would need to retrieve the full list of entities each time we forward a single page, the application might feel sluggish if the list was huge or the sorting and filtering was complicated. It might therefore be sensible to implement some sort of caching scheme.

There are a couple of things to consider though:

- Our CherryPy applications are multithreading, so we should be aware of that, especially when storing things in a cache, as we don't want threads to trash the shared cache.
- We have to devise some scheme to limit the number of cached items as resources are limited.

- We must overcome the limitations of the statelessness of the HTTP protocol: each time the client issues a request. This request should contain all necessary information to determine if we have something cached for him available and of course we have to understand that each request may be served by a different thread.

These requirements can be satisfied if we change the line in the `index()` that retrieves the matching IDs into the following few lines:

Chapter7/browse.py

```
            if not (next is None and previous is None
                                and first is None and last is None):
                  cacheid=self.iscached(cacheid,sortorder,pattern)
            else:
                  cacheid=None
            if cacheid is None:
                  ids = self.entity.listids(
                        pattern=pattern,sortorder=sortorder)
                  cacheid = self.storeincache(ids,sortorder,pattern)
            else:
                  ids = self.getfromcache(cacheid,sortorder,pattern)
                  if ids == None:
                        ids = self.entity.listids(
                              pattern=pattern,sortorder=sortorder)
                        cacheid = self.storeincache(ids,sortorder,
pattern)
```

Because we will store a unique `cacheid` in a hidden `<input>` element, it will be passed as an argument when the form is submitted. We use this `cachid` together with the `sortorder` and `pattern` arguments to check whether a previously retrieved list of IDs is present in the cache with the `iscached()` method. Passing the `sortorder` and `pattern` arguments will enable the `iscached()` method to determine if these are changed and invalidate the cache entry.

`iscached()` will return the `cacheid` if it exists in the cache or `None` if it doesn't. `iscached()` will also return `None` if the `cacheid` does exist but the `sortorder` or `pattern` arguments were changed.

Next, we check if the `cacheid` is `None`. This may seem redundant, but if `index()` was called for the first time (without arguments, that is) none of the submit button arguments would be present and we wouldn't have checked the cache.

This is intended: if we would, at a later point, revisit this list, we would want a fresh set of items, not some old cached ones. After all, the contents of the database might have changed.

If the `cacheid` is `None` we retrieve a fresh list of IDs and store it in the cache together with the `sortorder` and `pattern` arguments. The `storeincache()` method will return a freshly minted `cacheid` for us to store in the hidden `<input>` element.

If the `cacheid` was not `None`, we use the `getfromcache()` method to retrieve the list of IDs from the cache. We check the returned value because between our checking for the existence of the key in the cache and retrieving the associated data, the cache might have been purged, in which case, we still call the `listids()` method.

The implementation of the `iscached()`, `getfromcache()`, and `storeincache()` method takes care of all the thread safety issues:

Chapter7/browse.py

```
    def chash(self,cacheid,sortorder,pattern):
        return cacheid + '-' + hex(hash(str(sortorder))) + '-' +
hex(hash(str(pattern)))
    def iscached(self,cacheid,sortorder,pattern):
        h=self.chash(cacheid,sortorder,pattern)
        t=False
        with self.cachelock:
            t = h in self.cache
            if t :
                self.cache[h]=(time(),self.cache[h][1])
        return cacheid if t else None
    def cleancache(self):
        t={}
        with self.cachelock:
            t={v[0]:k for k,v in self.cache.items()}
        if len(t) == 0 :
            return
        limit  = time()
        oldest = limit
        limit -= 3600
        key=None
        for tt,k in t.items():
            if tt<limit:
                with self.cachelock:
                    del self.cache[k]
            else:
```

```
                        if tt<oldest:
                                oldest = tt
                                key = k
        if key:
                with self.cachelock:
                        del self.cache[key]

    def storeincache(self,ids,sortorder,pattern):
        cacheid=uuid().hex
        h=self.chash(cacheid,sortorder,pattern)
        if len(self.cache)>self.cachesize :
                self.cleancache()
        with self.cachelock:
                self.cache[h]=(time(),ids)
        return cacheid

    def getfromcache(self,cacheid,sortorder,pattern):
        ids=None
        h=self.chash(cacheid,sortorder,pattern)
        with self.cachelock:
                try:
                        ids=self.cache[h][1]
                except KeyError:
                        pass
        return ids
```

All methods use the `chash()` method to create a unique key from the `cacheid` and the `sortorder` and `pattern` arguments. `iscached()` waits until it acquires a lock to check if this unique value is present in the cache. If it is, it updates the associated value, a tuple consisting of a timestamp and a list of IDs. By updating this timestamp here, we reduce the chance that this item is purged from the cache between the check for existence and the actual retrieval.

The `getfromcache()` method creates a unique key with the `chash()` method in the same way `iscached()` did and waits to acquire the lock before it uses the key to retrieve the value from the cache. If this fails, a `KeyError` will be raised that will be caught, causing the `None` value to be returned as that was what the IDs variable was initialized to.

The `storeincache()` method first creates a new `cacheid` using one of the `uuid()` functions from Python's `uuid` module, essentially creating a random string of hexadecimal characters. Together with the `sortorder` and `pattern` arguments, this new `cacheid` is used to generate a unique key.

Before we store the list of IDs in the cache, we check whether there is any space left by comparing the number of keys in the cache to the maximum length we are prepared to accept. If there isn't any room left, we make room by calling the `cleancache()` method that will remove any entries that are too old. We then store the IDs together with a time stamp after acquiring a lock and return the `cacheid` just generated.

The final cog in our caching machinery is the `cleancache()` method. After requiring a lock, a reverse map is built, mapping timestamps to keys. If this map holds any items, we use it to locate any key that is older than an hour. Those are deleted after acquiring a lock.

The whole business with acquiring a lock and releasing it as quick as possible instead of acquiring the lock and doing all the cache-related business in one go ensures that other threads accessing the cache do not have to wait very long, which keeps the whole application responsive.

If the age of an entry is less than an hour, we keep notes to see which of the remaining ones is the oldest to remove that one at the end. This way, we ensure that we always retire at least one entry, even if there aren't any really old ones.

The books application revisited

With that much versatile code available, constructing a new lean and mean version of our books application becomes very straightforward.

Time for action – creating a books application, take two

Run the code in `books2.py` and point your web browser to `http://localhost:8080`.

After logging in (a default username/password combination of admin/admin will be present), you will be presented with a list of entities to browse (books and authors) and after clicking on **Books**, a screen will present itself that closely resembles the general Browse application (the page still has a Spartan look because no CSS is added at this point):

Thanks to some JavaScript goodness, our browse screen is embedded in the page instead of functioning standalone, yet all functionality is retained, including skipping forward and backward as well as sorting. New books or authors may be added by clicking the **Add new** button.

What just happened?

When we take a look at the code in `books2.py`, we see that its main part consists of definitions of entities, relations, and specific `Browse` entities that are combined together to form a CherryPy application:

Chapter7/books2.py

```
import os
import cherrypy

from entity import AbstractEntity, Attribute
from relation import AbstractRelation

from browse import Browse
from display import Display
from editor import Editor

from logondb import LogonDB

db="/tmp/book2.db"

class Entity(AbstractEntity):
    database = db

class Relation(AbstractRelation):
    database = db

class User(Entity):
    name = Attribute(notnull=True, unique=True,
           displayname="Name")

class Book(Entity):
    title = Attribute(notnull=True, displayname="Title")
    isbn  = Attribute(displayname="Isbn")
    published = Attribute(displayname="Published")

class Author(Entity):
    name = Attribute(notnull=True, unique=True,
           displayname="Name", primary=True)

class OwnerShip(Relation):
    a = User
    b = Book

class Writer(Relation):
    a = Book
```

```
    b = Author

logon = LogonDB()

class AuthorBrowser(Browse):
    display = Display(Author)
    edit = Display(Author, edit=True, logon=logon)
    add = Display(Author, add=True, logon=logon)

class BookBrowser(Browse):
    display = Display(Book)
    edit = Display(Book, edit=True, logon=logon,
           columns=Book.columns+[Author])
    add = Display(Book, add=True, logon=logon,
           columns=Book.columns+[Author])

with open('basepage.html') as f:
    basepage=f.read(-1)

class Root():
    logon = logon
    books = BookBrowser(Book,
           columns=['title','isbn','published',Author])
    authors = AuthorBrowser(Author)

    @cherrypy.expose
    def index(self):
           return Root.logon.index(returnpage='../entities')

    @cherrypy.expose
    def entities(self):
           username = self.logon.checkauth()
           if username is None :
                  raise HTTPRedirect('.')
           user=User.list(pattern=[('name',username)])
           if len(user) < 1 :
                  User(name=username)

           return basepage%'''<div class="navigation">
           <a href="books">Books</a>
           <a href="authors">Authors</a>
           </div><div class="content">
           </div>
           <script>
                  ... Javascript omitted ...
           </script>
           '''

cherrypy.engine.subscribe('start_thread',
    lambda thread_index: Root.logon.connect())
```

```
current_dir = os.path.dirname(os.path.abspath(__file__))
cherrypy.quickstart(Root(),config={
          '/':
          { 'log.access_file' :
                os.path.join(current_dir,"access.log"),
          'log.screen': False,
          'tools.sessions.on': True
          }
    })
```

After importing the modules we need, we define `User`, `Book`, and `Author` entities and an `OwnerShip` class, to define the relation between a book and a user. Likewise, we define a `Writer` class that defines the relation between a book and its author(s).

The next step is to create an instance of a `LogonDB` class (highlighted) that will be used in many parts of our CherryPy application to verify that the user is authenticated.

The bulk of the CherryPy application consists of two `Browse` classes, one for books and one for authors. Each class has display, edit, and add class variables that point to further branches of our application that are served by `Display` instances.

The `Root` class we define ties all of this together. It refers to the `LogonDb` instance created earlier in its `logon` class variable, and its `books` and `authors` class variables point to the previously defined `Browse` instances. It also defines an `index()` method that merely presents a logon screen if the user is not yet authenticated and if he/she is, redirects the user to the entities page. The `entities()` method which serves this page makes sure there is a corresponding user in the database (highlighted) and presents a base page consisting of a navigation `div` and a content `div` that will be filled when one of the links in the navigation section is clicked, and some JavaScript to tie everything together.

Before we examine the JavaScript, it might be good to take a look at the illustration to see how the application tree looks:

Path	Method
/	`Root.index()`
/entities	`Root.entities()`
/logon	`LogonDB.index()`
/books	`BooksBrowser.index()`
/add	`Display().index()`
/edit	`Display().index()`
/display	`Display().index()`
/authors	`AuthorBrowser.index()`
/add	`Display().index()`
/edit	`Display().index()`
/display	`Display().index()`

(Note that the `edit`, `add`, and `display` branches are each serviced by a different instance of `Display`).

Earlier, we saw that the `Browse` class we created was able to function standalone: clicking on any of the buttons referred to the same URL that served up the form in the first place. This setup makes it possible to use different `Browse` instances in different parts of an application tree, but here we want to replace a part of a page with the form produced by the `Browse` instance using an AJAX call. The problem then is that submitting a form without an `action` attribute will result in a request to the current URL, that is, the one referring to the page the form is embedded in, not the one that produces the form.

Fortunately, we can use jQuery to solve this problem by altering the `action` attributes of the freshly loaded forms to point to the URL that served those forms:

Chapter7/books2.py

```
$.ajaxSetup({cache:false,type:"GET"});
$(".navigation a").click(function (){
  var rel = $(this).attr('href');
  function shiftforms(){
    $(".content form").each(function(i,e){
    $(e).attr('action',
      rel+'/'+$(e).attr('action'));
      $('[type=submit]',e).bind('click',
        function(event){
        var f = $(this).parents('form');
        var n = $(this).attr('name');
```

```
            if (n != ''){
              n = '&'+n+'='+$(this).attr('value');}
            $(".content").load(f.attr('action'),
              f.serialize()+n,shiftforms);
            return false;
          });
        });
      };
      // change action attributes of form elements
      $(".content").load($(this).attr('href'),shiftforms);
      return false;
    });
```

This is accomplished by adding a click event handler to the links in the navigation area. That will not only prevent the default action but load the HTML produced by the URL referred to by the `href` attribute and pass a callback function that will alter the action attributes of any freshly loaded `<form>` elements (highlighted).

The `shiftforms()` function first prepends the original `href` contents to the `action` attributes and then binds a click handler to each button or input element with a `type` attribute equal to submit.

It would not be sufficient to add a submit handler to the form, because we don't want to let the `<form>` perform its default action. When a form is submitted, the contents of the page are replaced and this is not what we want. Instead, we want to replace the contents of the content `div` so we have to `load()` the URL from the form's `action` attribute ourselves.

This also means that we have to serialize the contents of the form to add as parameters to this URL, but jQuery's `serialize()` function will not serialize submit buttons. We, therefore, end up with adding a click handler to submit buttons in order to be able to determine the submit button that was clicked, so we can construct a complete list of parameters, including the name and value of the submit button.

Summary

We took a critical look at the framework we developed so far, and made improvements to the framework to make it more versatile and simpler to use for a developer.

Specifically, we covered:

- How to do away with explicit database and thread initialization.
- How to wield the awesome power of Python metaclasses to synchronize the creation of Python classes and their corresponding database tables.
- How to use those same metaclasses to alter the definitions of existing classes to create a much more intuitive interface when dealing with relations.
- How to implement a `Browse` class to navigate through large collections of entities in an efficient way using caches.
- How to rewrite the books application in a much simpler way with this reworked framework.

We have still glossed over several issues, including how to display and edit instances. In the last three chapters, we will develop a Customer Relationship Management application and fill in the final bits, including controlling how to restrict some actions to specific persons and how to allow for end user customization of the application.

8
Managing Customer Relations

There is more to an entity framework and CherryPy application code than just merely browsing lists. The user must be able to add new instances and edit existing ones.

In this chapter, we will:

- See how to display instances
- How to add and edit instances
- How to provide auto complete functionality to attributes referring to other entities
- How to implement picklists

So let's get on with it...

A critical review

Now that we have created an object relational framework in the form of an entity and relation modules, it is time for a critical reappraisal.

A couple of larger and smaller issues may hinder us in quickly prototyping and implementing a database-driven application:

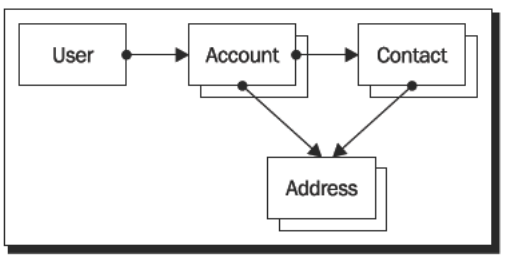
- We already keep an administration of the additional properties of the entity attributes, for example, whether an attribute has a validator function. It might be a good idea to store things like the preferred representation of an attribute's value as well. We also want to have the possibility of keeping a record of allowed values, so we can implement picklists

- Although the framework is flexible enough for a developer to quickly implement a database-driven application, it does not have any functionality to let an end user alter the database schema. It is not possible to add an attribute to an entity, for example. Even if this were possible, we would still need some authorization scheme to limit this functionality to authorized users only.

In the following chapters, we will tackle these limitations one-by-one and each step will bring us closer to implementing our final example: a customer relations management application. Some parts of this process require us to perform some pretty sophisticated Python tricks, but these parts are clearly flagged and may be skipped.

Designing a Customer Relationship Management application

Our first revision of CRM will start off with a bare bones implementation. It is about as simple as the books application and its data model is illustrated in the next diagram:



The web application will serve a single company and the users are typically the sales representatives and back office employees. In this basic form, an Account is the company we are interested in with a couple of attributes like name and the type of business. We also keep records of Contacts; these are people that may be associated with an Account. These Contacts have attributes like name, gender, and so on. Both Accounts and Contacts may have any number of addresses.

Time for action – implementing a basic CRM

Have a look at the following code (available as `crm1.py`). It will define the entities identified in the previous section and the result, when run, will have a familiar look:

We've added a little bit of CSS styling to order the elements on the page, but in the final revision, we will give it a much more attractive look. Clicking on the **Add new** button will allow you to add a new entity.

What just happened?

These humble beginnings in implementing CRM were accomplished by the code in `crm1.py`:

Chapter8/crm1.py

```
import os
import cherrypy

from entity import AbstractEntity, Attribute, Picklist,
AbstractRelation

from browse import Browse
from display import Display
from editor import Editor

from logondb import LogonDB

db="/tmp/crm1.db"

class Entity(AbstractEntity):
  database = db

class Relation(AbstractRelation):
  database = db

class User(Entity):
  name = Attribute(notnull=True, unique=True,
            displayname="Name", primary=True)

class Account(Entity):
  name = Attribute(notnull=True,
            displayname="Name", primary=True)

class Contact(Entity):
  firstname = Attribute(displayname="First Name")
  lastname  = Attribute(displayname="Last Name",
```

```
                notnull=True, primary=True)
    gender    = Attribute(displayname="Gender",
                notnull=True,
                validate=Picklist(
                        Male=1,
                        Female=2,
                        Unknown=0))
    telephone = Attribute(displayname="Telephone")

class Address(Entity):
    address   = Attribute(displayname="Address",
                notnull=True, primary=True)
    city      = Attribute(displayname="City")
    zipcode   = Attribute(displayname="Zip")
    country   = Attribute(displayname="Country")
    telephone = Attribute(displayname="Telephone")

class OwnerShip(Relation):
    a = User
    b = Account

class Contacts(Relation):
    a = Account
    b = Contact

class AccountAddress(Relation):
    a = Account
    b = Address

class ContactAddress(Relation):
    a = Contact
    b = Address
```

The first part is all about defining the entities and the relations between them according to the data model we sketched earlier. The concept is pretty much the same as for the books application, but for one important detail, the use of a **picklist** to limit the allowable choices for gender (highlighted). We will study these picklists in detail later in this chapter.

The next part creates the actual CherryPy application, with a `Browse` page for each entity (highlighted):

Chapter8/crm1.py

```
logon = LogonDB()

class AccountBrowser(Browse):
    display = Display(Account)
    edit = Display(Account, edit=True, logon=logon,
                                columns=Account.columns+[Address,User])
```

```
    add = Display(Account, add=True, logon=logon,
                                columns=Account.columns+[Address,User])
class UserBrowser(Browse):
    display = Display(User)
    edit = Display(User, edit=True, logon=logon)
    add = Display(User, add=True, logon=logon)
class ContactBrowser(Browse):
    display = Display(Contact)
    edit = Display(Contact, edit=True, logon=logon,
                                columns=Contact.
columns+[Account,Address])
    add = Display(Contact, add=True, logon=logon,
                                columns=Contact.
columns+[Account,Address])
class AddressBrowser(Browse):
    display = Display(Address)
    edit = Display(Address, edit=True, logon=logon)
    add = Display(Address, add=True, logon=logon)
```

The final part defines a `Root` class with an `index()` method that will force the user to identify himself/herself first (highlighted) and will then redirect the user to the `/entities` page, served by the `entities()` method.

This method will serve up a basepage with a navigation section that will allow the user to select a browse page for a type of entity and a content division which is initially empty, but will act as a container for either the chosen browse component or any edit or add page.

Chapter8/crm1.py

```
with open('basepage.html') as f:
    basepage=f.read(-1)
class Root():
    logon   = logon
    user    = UserBrowser(User)
    account = AccountBrowser(Account,
                        columns=Account.
columns+[User,Address,Contact])
    contact = ContactBrowser(Contact,
                        columns=Contact.columns+[Address,Account])
    address = AddressBrowser(Address)
    @cherrypy.expose
    def index(self):
          return Root.logon.index(returnpage='../entities')
    @cherrypy.expose
    def entities(self):
```

```
            username = self.logon.checkauth()
            if username is None :
                  raise HTTPRedirect('.')
            user=User.list(pattern=[('name',username)])
            if len(user) < 1 :
                  User(name=username)
            return basepage%'''
            <div class="navigation">
                  <a href="user">Users</a>
                  <a href="account">Accounts</a>
                  <a href="contact">Contacts</a>
                  <a href="address">Addresses</a>
            </div>
            <div class="content">
            </div>
            <script>
             ... Javascript omitted ...
            </script>
            '''
cherrypy.config.update({'server.thread_pool':1})
cherrypy.engine.subscribe('start_thread',
   lambda thread_index: Root.logon.connect())
current_dir = os.path.dirname(os.path.abspath(__file__))
cherrypy.quickstart(Root(),config={
          '/':
          { 'log.access_file' :
                os.path.join(current_dir,"access.log"),
          'log.screen': False,
          'tools.sessions.on': True
          },
          '/browse.js':
          { 'tools.staticfile.on':True,
          'tools.staticfile.filename':current_dir+"/browse.js"
          },
          '/base.css':
          { 'tools.staticfile.on':True,
          'tools.staticfile.filename':current_dir+"/base.css"
          }
   })
```

Adding and editing values

Until now, we did not look closely at the `Display` class, although it is used in various

incarnations within the application that we set up with CherryPy. The `Display` class combines a number of functions. It:

- Displays detailed values of an instance
- Allows those values to be edited
- Displays a form that allows us to add a completely new instance
- Processes the input from the edit and add forms

The reason to bundle these functions is twofold: displaying the labels and values for reading, editing, or adding an instance shares a lot of common logic, and by processing the results within the same class method, we can refer to the `action` attribute of a `<form>` element in a way that allows us to mount an instance of the `Display` class from anywhere in the application tree.

Time for action – adding an instance

To understand the `Display` class, let us create a very simple application. Type in the following code and run it:

Chapter8/crmcontact.py

```
import os
import cherrypy

from entity import AbstractEntity, Attribute, Picklist

from browse import Browse
from display import Display

from logondb import LogonDB

db="/tmp/crmcontact.db"

class Entity(AbstractEntity):
    database = db

class Contact(Entity):
    firstname = Attribute(displayname="First Name")
    lastname  = Attribute(displayname="Last Name",
                                          notnull=True,
primary=True)
    gender    = Attribute(displayname="Gender",
                                          notnull=True,
                                          validate=Picklist(

Male=1,Female=2,Unknown=0))
    telephone = Attribute(displayname="Telephone")
```

```
class ContactBrowser(Browse):
    edit = Display(Contact, edit=True)
    add  = Display(Contact, add=True)

current_dir = os.path.dirname(os.path.abspath(__file__))

cherrypy.quickstart(ContactBrowser(Contact),config={
          '/':
          { 'log.access_file' :
                os.path.join(current_dir,"access.log"),
          'log.screen': False,
          'tools.sessions.on': True
          }
})
```

When you point your browser to `http://localhost:8080`, you will be presented with an empty list of contacts that you may expand by clicking the **Add** button. This will present you with the following screen:

Here you may enter new values, and when you click the add button, a new contact will be added to the database, after which, you will return to the list of contacts, but now with an extra one added.

What just happened?

In the application tree that we constructed, we mounted several instances of the `Display` class, each with its own initialization parameters. These parameters are merely stored in the instance by the `__init__()` method for referral later:

Chapter8/display.py

```
    def __init__(self, entity, edit=False, add=False,
                               logon=None, columns=None):
          self.entity = entity
          self.edit = edit
          self.add = add
          self.logon = logon
          if columns is None:
                self.columns = entity.columns
          else:
                self.columns = columns
```

The most important parameter is `entity`. This will be the `Entity` class that we want `Display` to be able to add or edit.

`__init__()` also takes an `edit` or `add` parameter that when set will determine the type of activity this instance of `Display` will perform. If neither is given, an instance will just be displayed without the possibility of altering its attributes. In the stripped down `crmcontact.py` application, we created a `ContactBrowser` class that holds references to two different instances of the `Display` class. The one in the `add` class variable is created with an `add` attribute set to `True`, while the one in the `edit` variable is created with an `edit` attribute set to `True`. The **Add new** button in the browser is equipped with a click handler that will replace the browse list with the form that will be served by the `Display` instance that was created with the `add` argument.

Time for action – editing an instance

We also want to open a form that will allow the user to edit an existing instance when double-clicked in the browse list. In the stripped down application that we created in the previous section, we merely created the `ContactBrowser` class as a subclass of `Browse`. If we want to add an additional double-click handler to the browse list element, we will have to override the `index()` method.

In the definition of the `ContactBrowser` class, add the following to the definition of the `index()` method (the complete code is available as `crmcontactedit.py`):

Chapter8/crmcontactedit.py

```
    @cherrypy.expose
    def index(self, _=None,
           start=0, pattern=None, sortorder=None, cacheid=None,
           next=None, previous=None, first=None, last=None,
           clear=None):
           s="".join(super().index(_, start, pattern, sortorder,
                                   cacheid, next,previous, first, last,
clear))
           s+='''
           <script>
           $("table.entitylist tr").dblclick(function(){
                        var id=$(this).attr('id');
                        $("body").load('edit/?id='+id);
                 });
           </script>
           '''
           return basepage%s
```

The code merely gathers the output from the original `index()` method of the `Browse` class (highlighted) and adds a `<script>` element to it that will add a double-click handler to each `<tr>` element in the browse list. This click handler will replace the body with the form served by the edit URL, which will be passed an `id` parameter equal to the `id` attribute of the `<tr>` element.

If you run `crmcontactedit.py`, you will be presented with the same list of contacts as before and if it is empty, you may first need to add one or more contacts. Once these are present, you can double-click on any of them to be presented with an edit screen:

This looks very similar to the add screen, but changing values here and clicking the **Edit** button will alter instead of adding a contact and returning you to the list of contacts.

What just happened?

Let us have a look at how the `Display` class handles the editing of instances.

All interaction by an instance of the `Display` class is provided by a single method: `index()` (full code is available in `display.py`):

Chapter8/display.py

```
@cherrypy.expose
def index(self, id=None, _=None,
      add=None, edit=None, related=None, **kw):
  mount = cherrypy.request.path_info
  if not id is None :
    id = int(id)
  kv=[]
  submitbutton=""
  if edit or add:
    ... code to process the results of an edit/add form omitted
  action="display"
  autocomplete=''
  if not id is None:
    e=self.entity(id=id)
    for c in self.columns:
      if c in self.entity.columns:
        kv.append('<label for="%s">%s</label>'%
              (c,self.entity.displaynames[c]))
        if c in self.entity.validators and type(
            self.entity.validators[c])==Picklist:
```

```
            kv.append('<select name="%s">'%c)
            kv.extend(['<option %s>%s</option>'%
              ("selected" if v==getattr(e,c)
              else "",k)
            for k,v in self.entity.validators[c]
              .list.items()])
            kv.append('</select>')
          else:
          kv.append(
          '<input type="text" name="%s" value="%s">'%
            (c,getattr(e,c)))
        elif issubclass(c,AbstractEntity):
          kv.append(
          '<label for="%s">%s</label>'%
          (c.__name__,c.__name__))
          v=",".join([r.primary for r in e.get(c)])
          kv.append(
          '<input type="text" name="%s" value="%s">'%
           (c.__name__,v))
          autocomplete += '''
$("input[name=%s]").autocomplete({source:"%s",minLength:2});
          '''%(c.__name__,
             mount+'autocomplete?entity='+c.__name__)
      yield self.related_entities(e)
      if self.edit:
        action='edit'
        submitbutton='''
        <input type="hidden" name="id" value="%s">
        <input type="hidden" name="related" value="%s,%s">
        <input type="submit" name="edit" value="Edit">
        '''%(id,self.entity.__name__,id)
    elif self.add:
      action='add'
      for c in self.columns:
        if c in self.entity.columns:
          kv.append('<label for="%s">%s</label>'%(
            c,self.entity.displaynames[c]))
          if c in self.entity.validators and type(
              self.entity.validators[c])==Picklist:
            kv.append('<select name="%s">'%c)
            kv.extend(['<option>%s</option>'%v
              for v in self.entity.validators[c].
                list.keys()])
```

```
            kv.append('</select>')
          else:
            kv.append('<input type="text" name="%s">'
                  %c)
        elif c=="related":
          pass
        elif issubclass(c,AbstractEntity):
          kv.append('<label for="%s">%s</label>'%
            (c.__name__,c.__name__))
          kv.append('<input type="text" name="%s">'%
            c.__name__)
          autocomplete += '''
$("input[name=%s]").autocomplete({source:"%s",minLength:2});
          '''%(c.__name__,
             mount+'autocomplete?entity='+c.__name__)
      submitbutton='''
      <input type="hidden" name="related" value="%s">
      <input type="submit" name="add" value="Add">
      '''%related
    else:
      yield """cannot happen
      id=%s, edit=%s, add=%s, self.edit=%s, self.add=%s
      """%(id,edit,add,self.edit,self.add)
    yield '<form action="%s">'%action
    for k in kv:
      yield k
    yield submitbutton
    yield "</form>"
    yield '<script>'+autocomplete+'</script>'
```

Depending on the parameters passed to the index() method and the information stored when the `Display` instance was initialized, `index()` performs different but similar actions.

When called without the `add` or `edit` parameter, `index()` is called to display, edit, or add an instance and the first part of the code is skipped.

The `add` and `edit` parameters to `index()` are different from the ones passed to `__init__()`.

First, we check if the `id` parameter is present (highlighted). If not, we're expected to present an empty form to let the user enter the attributes for an all new instance. However, if an `id` parameter is present, we have to display a form with values.

To present such a form, we retrieve the entity with the given ID and check which columns we have to display and see if such a column is an attribute of the entity we are dealing with (highlighted). If so, we append to the `kv` list a `<label>` element with the display name of the column and an `<input>` or `<select>` element, depending on whether we are dealing with an unrestricted text field or a picklist. If we are dealing with a picklist, the available choices are added as `<option>` elements.

If the column to display is not an attribute of the entity but another entity class, we are dealing with a relation. Here we also add a `<label>` element and an `<input>` field, but we also add JavaScript code to the `autocomplete` variable that when executed will convert this `<input>` element into an autocomplete widget, which will retrieve its choices from the `autocomplete()` method in this same `Display` instance.

Only if this `Display` instance was initialized to perform the edit function (highlighted), we append a submit button and set the `action` variable to edit (the last part of the URL the values of the `<form>` element will be submitted to). We also add an extra hidden input element that holds the ID of the instance we are editing.

Constructing the empty form to add a new instance is almost the same exercise, only in this case, no values are filled in any of the `<input>` elements.

The final lines of code (highlighted) are shared again and used to deliver the `<form>` element based on the components just created for either an edit/display form or an empty add form together with any JavaScript generated to implement the autocomplete features. A typical sample of HTML generated for an edit form may look like this:

```
<form action="edit">
    <label for="firstname">First Name</label>
    <input name="firstname" value="Eva" type="text">
    <label for="lastname">Last Name</label>
    <input name="lastname" value="Johnson" type="text">
    <label for="gender">Gender</label>
    <select name="gender">
           <option selected="selected">Unknown</option>
           <option>Male</option>
           <option>Female</option>
    </select>
    <label for="telephone">Telephone</label>
    <input name="telephone" value="" type="text">
    <label for="Account">Account</label>
    <input name="Account" value="" type="text">
    <label for="Address">Address</label>
    <input name="Address" value="" type="text">
    <input name="id" value="2" type="hidden">
    <input name="edit" value="Edit" type="submit">
```

```
</form>
<script>
$("input[name=Account]").autocomplete({source:"autocomplete?entity=Acc
ount",minLength:2});
$("input[name=Address]").autocomplete({source:"autocomplete?entity=Add
ress",minLength:2});
</script>
```

If the `index()` method of `Display` is called with either the `add` or the `edit` parameter present (typically the result of clicking a submit button in a generated edit or add form), different code is executed:

Chapter8/display.py

```
@cherrypy.expose
def index(self, id=None, _=None,
      add=None, edit=None, related=None, **kw):
  mount = cherrypy.request.path_info
  if not id is None :
    id = int(id)
  kv=[]
  submitbutton=""
  if edit or add:
    if (edit and add):
      raise HTTPError(500)
    if not self.logon is None:
      username=self.logon.checkauth()
      if username is None:
        raise HTTPRedirect('/')
    if add:
      attr={}
      cols={}
      relations={c.__name__:c for c in self.columns
            if type(c)!=str}
      for k,v in kw.items():
        if not k in self.entity.columns:
          attr[k]=v
          if not k in relations :
            raise KeyError(k,
                  'not a defined relation')
        else:
          cols[k]=v
      e=self.entity(**cols)
      for k,v in attr.items():
        if v != None and v != '':
```

```
            relentity = relations[k]
            primary = relentity.primaryname
            rels = relentity.listids(
              pattern=[(primary,v)])
            if len(rels):
              r = relentity(id=rels[0])
            else:
              r = relentity(**{primary:v})
            e.add(r)
        if not related is None and related != '':
          r=related.split(',')
          re=e.relclass[r[0]](id=int(r[1]))
          e.add(re)
        redit = sub(Display.finaladd,'',mount)
        raise cherrypy.HTTPRedirect(redit)
      elif edit:
        e=self.entity(id=id)
        e.update(**kw)
        raise cherrypy.HTTPRedirect(
          mount.replace('edit','').replace('//','/'))
          ...code to display and instance omitted
```

Only one of `edit` or `add` should be present; if both are present we raise an exception. If the user is not authenticated, editing an instance or adding a new one is not allowed, and we unceremoniously redirect him/her to the homepage (highlighted).

If the `add` parameter is present, we will be creating a brand new instance. The first item of order is to check all incoming parameters to see if they are either an attribute of the entity that we will be creating (highlighted) or the name of a related entity. If not, an exception is raised.

The next step is to create the new entity (highlighted) and establish any relations.

Adding relations

One of the things we silently glossed over in the previous sections was the functionality to define relations between entities. Sure, the implementation of the `Display` class did allow for the creation of new instances, but we did not address how to define a relation, even though `Display` is already perfectly capable of showing columns that point to related entities like authors.

We could have hardcoded this behavior into specific implementations of `Display` like we did earlier when we implemented the first version of the books application, but this doesn't play well with the idea of creating components that can figure out those things for themselves, leaving the developer of the web application with fewer things to worry about.

The previous incarnation of the `relation` module was not quite up to this: we could define and administer a relation all right, but we'd have to do that by referring explicitly to an instance of a `Relation` class.

Because this isn't intuitive, we created a second version of the `relation` module that allows us to use the `add()` method inserted into the class definition of an `Entity` by the metaclass that creates a new relation. We do not have to care about the details: if we use `add()` to establish a relation between two entities, this is all taken care of.

This means that we can complete the add functionality of the `Display` class. For each column that refers to another entity (for example, the `Author` column of a book), we now implement some way for the user to make a choice, for example, with the autocomplete functionality, and process this choice in a rather simple manner: if it is empty, we do not add a relation, if it refers to a existing entity, add the relation and if not, create the related entity first before adding it.

We now have the functionality to refer to existing related items by their primary attribute or define a new one. However, for the end user, it might be very convenient to have auto completion on input fields that refer to related entities. This not only may save time, it also prevents inadvertently adding new entities when a typing error is made.

In previous chapters, we already encountered auto completion with the help of jQuery UI's autocomplete widget and we implemented the server-side functionality to retrieve lists of possible completions. All we have to do now is to make this functionality available in a manner that is independent from the actual related entity:

Chapter8/display.py

```
    @cherrypy.expose
    def autocomplete(self, entity, term, _=None):
           entity={c.__name__:c for c in self.columns
                  if type(c)!=str}[entity]
           names=entity.getcolumnvalues(entity.primaryname)
           pat=compile(term,IGNORECASE)
           return json.dumps(list(takewhile(lambda x:pat.match(x),
                                dropwhile(lambda x:not pat.
match(x),names))))
```

The HTML and JavaScript that is generated by the `index()` method of the `Display` class will ensure that the preceding `autocomplete()` method will be called with the name of the entity of which we want to retrieve column values.

Any related class that the instance we are editing refers to is stored in the `self.columns` instance variable, just like the names of the regular attributes. The highlighted line, therefore, collects those column names that are actually classes and creates a dictionary indexed by name, which holds the corresponding classes as values.

When we use the name of the related entity passed to the `autocomplete()` method as the index, we will get hold of the class. This class is used in the next line to retrieve all column values for the column marked as the primary column. The final code to return a JSON encoded list of all those values that start with the term argument is the same as implemented earlier.

Dictionary comprehensions are a new addition to Python 3.x, so it might be enlightening to write out the highlighted line in the example code in a more traditional manner:

```
classmap = {}
for c in self.columns:
    if type(c)!=str:
            classmap[c.__name__] = c
entity = classmap[entity]
```

Picklists

When we examined the code to generate a form to edit an instance, we did not look into the details of implementing picklists. Picklists are a great way to reduce input errors. Anywhere a limited list of values is permitted, we can use a picklist, thereby preventing the user from inadvertently entering a value that is not allowed. In doing so, we can also associate each possible value with a meaningful label.

We already have the possibility to add a validation function, but this function only checks the input; it does not provide us with a list of possible choices.

Time for action – implementing picklists

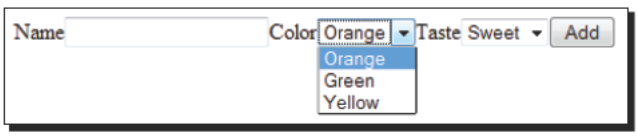
What we need is a way to indicate that an entity attribute is a picklist. Run the following code (available as `fruit.py`) and point your browser to `http://localhost:8080`.

Chapter8/fruit.py

```
import os
import cherrypy

from entity import AbstractEntity, Attribute, Picklist

from browse import Browse
from display import Display

from logondb import LogonDB

db="/tmp/fruits.db"
```

```
class Entity(AbstractEntity):
    database = db

class Fruit(Entity):
    name  = Attribute(displayname="Name")
    color = Attribute(displayname="Color",
           notnull = True,
           validate= Picklist([('Yellow',1),('Green',2),('Orange',0)]))
    taste = Attribute(displayname="Taste",
           notnull = True,
           validate= Picklist(Sweet=1,Sour=2))

class FruitBrowser(Browse):
    edit = Display(Fruit, edit=True)
    add  = Display(Fruit, add=True)

current_dir = os.path.dirname(os.path.abspath(__file__))

cherrypy.quickstart(FruitBrowser(Fruit),config={
           '/':
           { 'log.access_file' : os.path.join(current_dir,"access.
log"),
           'log.screen': False,
           'tools.sessions.on': True
           }
})
```

Click the **Add** button to create a new fruit instance. The `color` and `taste` attributes are defined as picklists, and clicking on the **Color** attribute, for example, may look like this:



What just happened?

In the `entity.py` file, we added a `Picklist` class to store the available choices and their values:

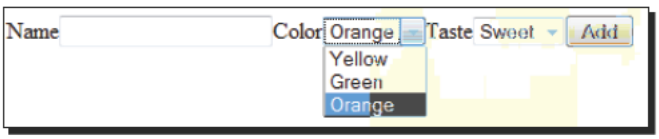
Chapter8/entity.py

```
class Picklist:
    def __init__(self,list=None,**kw):
           self.list = collections.OrderedDict(list) if not list is
None else collections.OrderedDict()
           self.list.update(kw)

    def __getitem__(self,key):
```

```
        return self.list[key]
```

The `Picklist` class is primarily a container for an `OrderedDict` (highlighted) and may be initialized either by a list or by passing any number of keywords to the `__init__()` method. However, the order of these keywords is not preserved, so even though we defined the color attribute of the fruit entity with this validate argument `validate= Picklist(Yellow=1, Green=2,Orange=0)`, the order in the drop-down box was **Orange**, **Green**, and **Yellow**.

So although convenient, passing keywords makes the use of an `OrderedDict` rather pointless. Therefore, the `__init__()` method also accepts a list of tuples of key/value pairs and if present, uses this list to initialize the dictionary. Now if we would use `validate= Picklist([('Yellow',1),('Green',2),('Orange',0)])`, the order would be preserved, as shown in the following screenshot. It has the added benefit of allowing us to specify any string as a key and not just strings that are valid Python identifiers.



We already saw in the `index()` method of the `Display` class how to retrieve a list of possible choices. An `Entity` itself also needs to know how to deal with attributes that are picklists, for example, when it updates such an attribute. The `__setattr__()` method of the `AbstractEntity` class will have to be adapted as follows:

Chapter8/entity.py

```
def __setattr__(self,name,value):
        if name in self.validators :
            if type(self.validators[name])==Picklist:
                try:
                    value=self.validators[name].list[value]
                except:
                    # key not known, try value directly
                    if not value in list(
                      self.validators[name].list.values()):
                        raise AttributeError(
            "assignment to "+name+" fails, "+
            str(value)+" not in picklist")
            elif not self.validators[name](value):
                raise AttributeError(
                "assignment to "+name+" does not validate")
        object.__setattr__(self,name,value)
```

The added lines (highlighted) check whether any validator is a `Picklist`, and if it is, tries to retrieve the value associated with the key. If this fails, it checks if the value that is entered is one of the values allowed. This way, it is valid to update a picklist attribute both with a key as well as a value. Given a `fruit` instance of the `Fruit` class defined earlier, the following lines are equivalent:

```
fruit.color = 'Green'
fruit.color = 2
```

Summary

We learned a lot in this chapter about how to present the end user with forms to manipulate instances in a way that does not need any hardcoded information.

Specifically, we covered how to display instances, add, and edit them, how to provide autocomplete functionality to attributes referring to other entities, and how to implement picklists.

All these items helped us to design and implement the first revision of a CRM application. Of course, there is more to a CRM application than just Accounts and Contacts and that is what we will look into in the next chapter.

9

Creating Full-Fledged Webapps: Implementing Instances

The framework for the rapid development of Python web applications is coming along nicely, but some distinct features are still lacking.

In this chapter, we take a look at some of them, notably:

- How to implement more complex relations
- How to create the necessary user interface components to maintain those relations
- And how to allow for a more fine-grained control of who is allowed to do what

These are some interesting challenges, so let's get started.

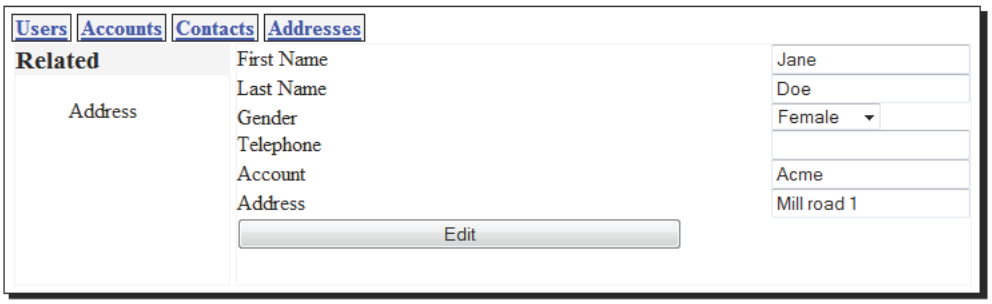
Even more relations

As we have seen, it was not very difficult to make `Display` capable of handling references to related entities. These relations, however, are limited to lookup relations (also known as many-to-one relations). A `Contact` entity, for example, refers to, at most, a single `Account` entity and the `Display` class allows us to select an `Account` when we edit a `Contact`.

But what do we need for the opposite situation? An `Account` may have many `Contacts` and both `Account` and `Contact` may have many `Addresses`. What we need is a way to make `Display` show which one-to-many relationships exist for an entity and provide the means to show those entities when the user clicks on such a relation.

Time for action – showing one-to-many relationships

The illustration shows what we might expect:



We have selected a `Contact` and its details are available for editing, including a reference to an `Account`. Now on the left, however, we have a sidebar showing the available one-to-many relations, in this case, the only one-to-many relation applicable to a `Contact` is `Address`.

What just happened?

To show a list of entities, we already have a suitable building block, the `Browse` class that will not only let us browse a list of entities in various ways, but is also capable of filtering those entities. In this example, we would want to show just those addresses that are associated with this specific contact.

We therefore add a new method to the `Display` class that will produce an HTML fragment together with some JavaScript to show the list of available one-to-many relations:

Chapter9/display.py

```
@staticmethod
def related_link(re,e):
  return '<li id="%s" class="%s" ref="%s">%s</li>'%(
    e.id,e.__class__.__name__,re.lower(),re)
def related_entities(self,e):
  r=['<div class="related_entities"><h3>Related</h3><ul>']
  if hasattr(e.__class__,'reltype'):
    r.extend([self.related_link(re,e)
      for re,rt in e.__class__.reltype.items()
      if (rt == '1:N' or rt == 'N:N')])
  r.append('</ul></div>')
  r.append('''
  <script>
```

```
        $('div.related_entities li').click(function(){
          var rel=$(this).attr("ref");
          var related=$("input[name=related]").val();
          $(".content").load(rel,
            $.param({
              "pattern" : $(this).attr("class") +
                    "," + $(this).attr("id"),
              "related": related}),
            function(){shiftforms(rel)});
        });
      </script>''')
      return "\n".join(r)
```

To determine which relations are available, `related_entities()` refers to the `reltype` class variable (highlighted), which is a dictionary of entity names and their type maintained by the `MetaRelation` class when a new relation is defined. For each suitable relation, a `<li>` element is produced with the help of the `related_link()` method.

These `<li>` elements have an `id` attribute that holds the unique ID of the referring entity (the ID of the contact in this example) and a `class` attribute that indicates the type of the referring entity (`Contact` in this case). The `<li>` elements also have a `rel` attribute that points to the URL that is serviced by a `Browse` class. For now, we derive this URL from the name of the entities we are referring to (in this case, address).

The final piece of HTML produced is a `<script>` element that installs an event handler on the `<li>` elements. This click handler will take the `ref` attribute of its associated `<li>` element to construct a URL that is subsequently used to open a new window. We will have to adapt the `index()` methods of the `Display` and `Browse` classes slightly to pass and process those attributes around, as we will see in a minute.

In our example, the resulting HTML fragment (minus the script element) would look like this:

```
<div class="related_entities">
    <h3>Related</h3>
    <ul>
        <li ref="address" class="Contact" id="1">Address</li>
    </ul>
</div>
```

And the `load()` call that will replace the contents of the `<div>` element with the `content` class would be passed the following URL, for example: `http://127.0.0.1:8080/address/?_=1295184609212&pattern=Contact,1&related=Contact,1`.

Note that we use jQuery's `param()` function here to convert an object containing several attributes to a string suitable to add to a URL. We could have simply passed the object here, but that would result in a POST action even though we configured all AJAX calls to use the HTTP GET method. Normally, that wouldn't be a problem, but if the final slash is missing in the URL, CherryPy will redirect us to the URL with the slash at the end and the AJAX call will be made again, but this time without the parameters appended! To prevent this possible awkwardness and to aid in debugging, we force the use of the GET method by constructing the full URL ourselves with the help of the `param()` function.

Time for action – adapting MetaRelation

In the `Display` class, we used the information about the type of relation stored by the `MetaRelation` metaclass. This is necessary because in recognizing that there is more than one type of relation, we need some way to indicate that when we define a new relation and act upon that information when creating a new class. Look at the following example code:

```
class A(Entity):
    pass
class B(Entity):
    pass
class AhasmanyB(Relation):
    a=A
    b=B
```

Here we express the relation between `A` and `B` to be one-to-many. If we would like to express the notion that an instance of `A` may refer only to a single `B` instance, we need some way to indicate that in the definition. One way of doing so is by reversing the assignments in the class definition of the variable:

```
class AreferstoasingleB(Relation):
          a=B
          b=A
```

The `MetaRelation` metaclass we defined earlier could act on such a definition as we arranged for the class dictionary of the relation being defined to be an `OrderedDict`, so in principle, we can act on the order of the definitions.

A slightly more explicit way of defining this is often clearer, so instead we opt for a `relation_type` attribute that can be assigned a string with the type of the relation. For example:

```
class AhasmanyB(Relation):
    a=A
    b=B
```

```
    relation_type='1:N'
class AreferstoasingleB(Relation):
    a=A
    b=B
    relation_type='N:1'
```

If we leave out the `relation_type`, a one-to-many relation is assumed.

What just happened?

Let's have a look at the changes and additions to `MetaRelation` needed to implement those semantics. We need two changes. The first is in the definition of the bridge table we use to administer the relation. We need an additional `unique` constraint here to enforce that in a one-to-many relation, the IDs in the column referring to the many side of the equation are unique.

This may sound counterintuitive, but let's say we have the following cars: a Volvo, a Renault, a Ford, and a Nissan. There are also two owners, John and Jill. Jill owns the Volvo and the Renault, and John the other cars. The tables might look like this:

Car	
ID	**make**
1	Volvo
2	Renault
3	Ford
4	Nissan

Owner	
ID	**name**
1	Jill
2	John

The table that reflects the ownership of the cars might look like this:

Ownership	
Car	**owner**
1	1
2	1
3	2
4	2

We see that while a *single* owner may have *many* cars, it is the numbers in the `Car` column that are unique because of this relation.

In order to define a table with those additional uniqueness constraints and to make the information about the type of relation available in the classes that form both halves of a relation, we have to adapt the final part of the `__new__()` method in the `MetaRelation` metaclass:

Chapter9/entity.py

```
if relationdefinition or '_meta' in classdict:
        a = classdict['a']
        b = classdict['b']
        r = '1:N'0
        if 'relation_type' in classdict: r = classdict['relation_type']
        if not r in ('N:1','1:N'): raise KeyError("unknown relation_
type %s"%r)
        classdict['relation_type'] = r

        if not issubclass(a,AbstractEntity) : raise TypeError('a not
an AbstractEntity')
        if not issubclass(a,AbstractEntity) : raise TypeError('b not
an AbstractEntity')
        runique = ' ,unique(%s_id)'%a.__name__
        if r == '1:N' : runique = ' ,unique(%s_id)'%b.__name__

        sql = 'create table if not exists %(rel)s ( %(a)
s_id references %(a)s on delete cascade, %(b)s_id references
%(b)s on delete cascade, unique(%(a)s_id,%(b)s_id)%(ru)
s)'%{'rel':classname,'a':a.__name__,'b':b.__name__,'ru':runique}

conn = sqlite.connect(classdict['_database'])
        conn.execute(sql)

        setattr(a,'get'+b.__name__,lambda self:getclass(self,b,
classname))
        setattr(a,'get',get)
        setattr(b,'get'+a.__name__,lambda self:getclass(self,a,
classname))
        setattr(b,'get',get)
        setattr(a,'add'+b.__name__,lambda self,entity:addclass(self,
entity,b,classname))
        setattr(a,'add',add)
        setattr(b,'add'+a.__name__,lambda self,entity:addclass(self,
entity,a,classname))
        setattr(b,'add',add)

        reltypes = getattr(a,'reltype',{})
        reltypes[b.__name__]=r
```

```
            setattr(a,'reltype',reltypes)
            reltypes = getattr(b,'reltype',{})
            reltypes[a.__name__]={'1:N':'N:1','N:N':'N:N','N:1':'1:N'}[r]
            setattr(b,'reltype',reltypes)

            relclasses = getattr(a,'relclass',{})
            relclasses[b.__name__]=b
            setattr(a,'relclass',relclasses)
            relclasses = getattr(b,'relclass',{})
            relclasses[a.__name__]=a
            setattr(b,'relclass',relclasses)

            joins = getattr(a,'joins',{})
            joins[b.__name__]=classname
            setattr(a,'joins',joins)
            joins = getattr(b,'joins',{})
            joins[a.__name__]=classname
            setattr(b,'joins',joins)
        return type.__new__(metaclass,classname,baseclasses,classdict)
```

The highlighted lines are the ones we added. The first set makes sure there is a `relation_type` attribute defined and if not, creates one with a default `'1:N'` value.

The second set of highlighted lines determines which column in the bridge table should receive an additional unique constraint and constructs the SQL query to create the table.

The final block of highlighted lines adds class attributes to both classes in the relation. All those attributes are dictionaries indexed by the name of an entity. The `reltype` attribute holds the type of the relation, so in a `Car` entity, we might obtain the type of relation with an `Owner` like this:

```
Car.reltype('Owner')
```

Which, if defined as in our previous example, will yield `'N:1'` (one or more cars may have a single owner).

Likewise, we can get information about the same relation from the perspective of the owner:

```
Owner.reltype('Car')
```

Which will yield the inverse, `'1:N'` (an owner may have more than one car).

Time for action – enhancing Display

What do we have to change to add the functionality to the `Display` class to pop up a list of items when the user clicks on a related tag?

- Because all activities of the `Display` class are served by its `index()` method, we will have to apply some changes there.
- The `index()` method both displays forms and processes the results when the submit button is pressed, so we have to look at both the aspects of the edit and add functionality.
- When an edit form is shown, this will always be initiated from double-clicking in a list of items shown by a `Browse` instance and will therefore be passed a related argument. This argument must be passed along with the contents of the form when the submit button is clicked in order to associate it with the item that initiated this edit action.

These issues require that we apply a few changes to the `index()` method.

What just happened?

The first thing we have to do is add a `related` parameter to the `index()` method. This parameter may hold the name of the entity and the ID of the specific related instance separated by a comma:

Chapter9/display.py

```
@cherrypy.expose
    def index(self,id=None,_=None,add=None,edit=None,related=None,**
kw):
```

When processing the information passed to the `index()` method of the `Display` class, the portion dealing with the results of an add action has to act on the information in the `related` parameter:

Chapter9/display.py

```
if not related is None and related != '':
    r=related.split(',')
    re=e.relclass[r[0]](id=int(r[1]))
    e.add(re)
```

If the method was called as the result of clicking the add button in a list of items, the `related` parameter will be non empty and we split it on the comma to retrieve the name of the entity and its ID.

The name of the entity is used to retrieve its class that was added to the `relclass` dictionary when the relation was defined and this class' constructor is called with the ID of the instance to create an object (highlighted). The relation between the item we are currently editing and the related item is subsequently established by the `add()` method.

Likewise, the portion responsible for delivering the add or edit form in the first place must include a hidden `<input>` element that holds the contents of the related parameter passed to it when the user clicked the add button in a page delivered by a `Browse` instance:

Chapter9/display.py

```
                    submitbutton='<input type="hidden" name="related"
value="%s"><input type="submit" name="add" value="Add">'%related
```

Time for action – enhancing Browse

All this passing around of the related parameter originates with the user clicking an 'add' button in a list of entities that, in its turn, was shown in response to clicking a related tag when editing or viewing an item.

Those lists of entities are generated by the `index()` method of the `Browse` class, so we have to make sure that suitable information (that is, the name of the entity that is listed together with the ID of the instance) is passed on.

This means we have to:

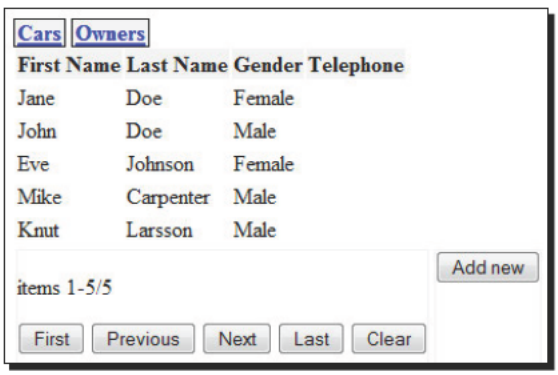
- Enhance the `index()` method to receive a related parameter that can be passed on when the 'add' button is clicked.
- Extend the code that generated the form associated with this add button with a hidden `<input>` element to hold this information, so that it may be passed on again to the `index()` method of the `Display` class.

If it sounds a little confusing how `Display` and `Browse` are connected, it may help to envision the following scenario:

- The user starts looking at a list of owners from the main menu and double-clicks a certain owner. This will result in an 'edit' form delivered by the `Display` class. Because double-clicking on an item will not pass a related argument, this argument in the `index()` method of `Display` will receive its default value of `None`.
- The edit form shows the details of the owner in the sidebar labeled '**Related**', we see a '**Cars**'entry.
- When the user clicks this **Cars** entry to show the list of cars related to this owner, this will result in the `index()` method of a `Browse` instance for the `Car` entity to be called with both a `related` and a `pattern` argument of `Owner,5`, for example.

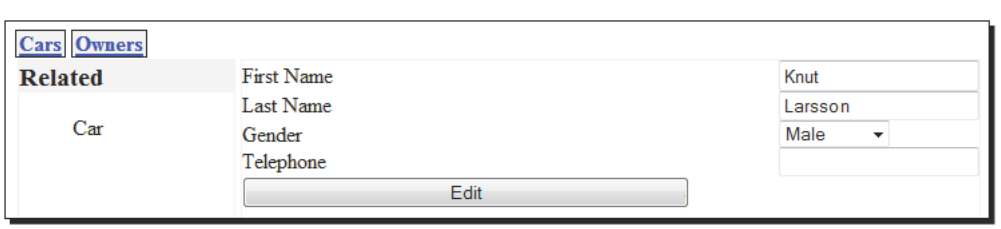
- This will result in a list of cars of the indicated owner and when the 'add' button in this list is clicked, it is again the `index()` method of the `Display` class that is called, but this time, a `Display` instance associated with the `Car` entity. It will be passed the `related` argument of `Owner,5`.
- Finally, when the user has entered the new car details and clicks 'add', the same `index()` method of the `Display` class is called, again with a `related` argument of `Owner,5` but also with an `add` argument. The car details will be used to create a new `Car` instance and the `related` argument to identify the `Owner` instance and associate the new car instance with.

The following series of screenshots illustrates what is happening. We start with a list of owners:



When we double-click on **Knut Larsson**, the following URL is sent to the server: `http://127.0.0.1:8080/owner/edit/?id=5&_=1295438048592` (the `id=5` indicates the instance, the number at the end is what jQuery adds to an AJAX call to prevent caching by the web browser).

The result will be an edit form for **Knut**:



A click on **Car** will result in the following URL being sent to the server: `http://127.0.0.1:8080/car/?_=1295438364192&pattern=Owner,5&related=Owner,5`.

We recognize the `related` and `pattern` arguments of `Owner,5` (that is, referring to **Knut**). Note that the commas in the arguments appended to this URL would be sent to the server encoded as `%2C`.

Why do we send both a `related` argument and a `pattern` argument containing the same information? For adding an entity to another entity, this is indeed redundant but if we would like to add the ability to transfer ownership as well as add a new entity, we would like to filter those cars that belong to some other owner and therefore we need to separately provide the `pattern` and `related` arguments.

If this is the first time we will be adding a car to **Knut**, the list of related cars will be empty:

If we now click on the **Add new** button, the following URL is constructed:

`http://127.0.0.1:8080/car/add/?_=1295438711800&related=Owner,5`, which will result in an add form for a new car:

After filling in the details, clicking on the **Add** button will result in a new car instance that will be associated with **Knut** even if we leave the **Owner** field empty because of the related argument passed yet again in the URL:

`http://127.0.0.1:8080/car/add/?_=1295439571424&make=Volvo&model=C30&color=Green&license=124-abc&Owner=&related=Owner,5&add=Add`.

What just happened?

To allow a `Browse` instance to receive and pass on a `related` attribute in the same manner as a `Display` instance, we need to make a few small changes. First, we have to alter the signature of the `index()` method:

Chapter9/browse.py

```
@cherrypy.expose
    def index(self, _=None, start=0, pattern=None, sortorder=None,
cacheid=None, next=None,previous=None, first=None, last=None,
clear=None, related=None):
```

All that is left then is to make sure that clicking the **Add** new button will pass on this value by including a hidden `<input>` element to the form:

Chapter9/browse.py

```
yield '<form method="GET" action="add">'
          yield '<input name="related" type="hidden"
value="%s">'%related
          yield '<button type="submit">Add new</button>'
          yield '</form>'
```

Access control

In the applications we designed so far, we took a very simple approach to access control. Based on someone's login credentials we either allowed access or not. We slightly expanded this notion in the books applications where deleting a book meant that the association between a book and an owner was deleted rather that removing the book instance from the database altogether.

In many situations, a finer control of privileges is required, but if this control is hardcoded into the application maintaining it will rapidly become unwieldy. We therefore need something that will allow us to manage access privileges in a simple way and in a manner that allows easy expansion.

Consider the following scenario: In a company using our CRM application, different accounts are owned by different sales persons. It's a small firm so everybody is allowed to see all the information on all the accounts but changing information for an account is restricted to the sales person that owns that account. Of course the sales manager, their boss, is allowed to change this information also, regardless of whether he owns an account or not.

We could implement such a strategy by letting the `update()` method of an `Entity` check whether this entity has an account and owned by the person doing the update and if not, whether the person is the sales manager.

Time for action – implementing access control

This scenario is implemented in `access1.py`:

Note: In the code distributed with this chapter and the following one, the `logon` class is not only initialized with an `admin` user (with `admin` as a password) but with the following three name/password combinations: `eve/eve`, `john/john`, and `mike/mike`.

If your run this application and point your browser to `http://localhost:8080`, you are presented with a list of accounts. If you have logged in as either **john** or **mike**—both sales persons—you can only alter the accounts owned by each of them. If however, you log in as eve, the sales manager, you can alter the information in all accounts.

What just happened?

The application is simple enough and follows a familiar pattern. The relevant definitions are shown here:

Chapter9/access1.py

```
from permissions1 import isallowed
class Entity(AbstractEntity):
    database = db

    def update(self,**kw):
           if isallowed('update', self, logon.checkauth(),
                                        self.getUser()):
                  super().update(**kw)
```

```
class Relation(AbstractRelation):
    database = db

class User(Entity):
    name = Attribute(notnull=True, unique=True,
           displayname="Name", primary=True)

class Account(Entity):
    name = Attribute(notnull=True, displayname="Name",
           primary=True)

class OwnerShip(Relation):
    a = User
    b = Account

class AccountBrowser(Browse):
    edit = Display(Account, edit=True, logon=logon,
           columns=Account.columns+[User])
    add  = Display(Account, add=True, logon=logon,
           columns=Account.columns+[User])

class UserBrowser(Browse):
    edit = Display(User, edit=True, logon=logon)
    add = Display(User, add=True, logon=logon)
```

The database distributed with the code (`access1.db`) contains a number of accounts already so the code does not contain any lines to create those. The important part is highlighted in the preceding code: it imports a `permissions1` module that contains a dictionary of permissions. This dictionary lists for each combination of entity, action, ownership, and username whether this is permissible or not.

We can now override the `update()` method in the `AbstractEntity` class (highlighted): We retrieve the username from the current sessions by calling the `checkauth()` method and pass it along to the `isallowed()` function, together with the name of the action we want to check (update in this case), the entity, and a list of users (the owners). If this checks out okay, we call the original `update()` method.

If we take a look at `permissions1.py`, we see that because, in this example, we only consider the `Account` entity and the update action in this list is quite small:

Chapter9/permissions1.py

```
import entity1

allowed = {
    'Account' : {
           'create' : {
                  'admin' : 'all',
                  'eve'  : 'all',
```

```
                'john' : 'owner',
                'mike'  : 'owner'
        },
        'update' : {
                'admin' : 'all',
                'eve'  : 'all',
                'john' : 'owner',
                'mike'  : 'owner'
        },
        'delete' : {
                'admin' : 'all',
                'eve'  : 'all',
        }
    }
}
def isallowed(action,entity,user,owner):
    if len(owner) < 1 : return True
    try:
            privileges = allowed[entity.__class__.__name__][action]
            if not user in privileges :
                    return False
            elif privileges[user] == 'all':
                    return True
            elif privileges[user] == 'owner' and user == owner[0].name:
                    return True
            else:
                    return False
    except KeyError:
            return True
```

The dictionary with privileges itself is called `allowed` (highlighted) and `permissions1.py` also defines a function called `isallowed()`, that will return `True` if there aren't any owners for this entity. Then it checks if there are any privileges known for this entity and action. If this is not the case, any exception will be raised because either the key for the entity or the key for the action does not exist.

If there are privileges known, we check if the user has specific privileges. If there is no key for the user, we return `False`. If there is, and the privilege is `all`, we return `True`: he/she may perform the action on this entity even for an entity instance he/she doesn't own. If the privilege is the owner, we only return `True` if the user is in fact the owner.

The aforementioned approach outlined is cumbersome for various reasons:

- If we would like to add a new salesperson, for example, we would have to add permission entries for each entity/action combination. In the example, we only considered the `Account` entity and the `update` action, but in a somewhat more realistic application, there would be tens of entities (like `Contact`, `Address`, `Quote`, `Lead`, and so on) and quite a few actions more to consider (for example, `delete` and `create`, but also actions that involve other entities like *changing* ownership or adding an address to an account). Also, if that sales person was promoted to sales manager, we would have to repeat the whole exercise again.
- If we added a new type of entity, we would have to add lines for each and every person in the company.
- Administering permissions in a Python module is not something you normally would expect a non-technical person to do as it is cumbersome, error prone, and requires the application to be restarted if something changes.

The last reason is why we will implement the list of permissions in the database. After all, we already have a framework that allows for easy manipulation of database entries with a web interface. The other reasons are why we will reconsider our first approach and will implement a scheme called **role-based access control**.

Role-based access control

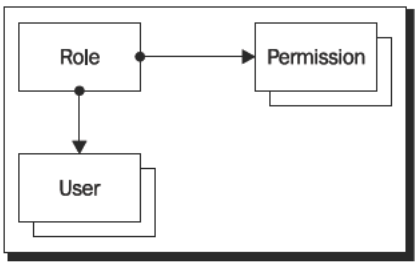
The idea in role-based access control is to assign one or more roles to people instead of specific permissions.

Permissions are then associated with a role, and if a person has more than one role, the permissions are merged. If a new person needs a set of permissions to use an application, or if a person's role in the organization changes, only the list of roles associated with that person needs to be changed instead of having to change the permissions for this person for each type of entity in the application.

Likewise, if we would extend the available types of entities, we would only have to define what permissions associated with a role (or roles) would apply to this new entity instead of defining this for every person.

A good starter for reading more about this is this Wikipedia article: `http://en.wikipedia.org/wiki/Role-based_access_control`.

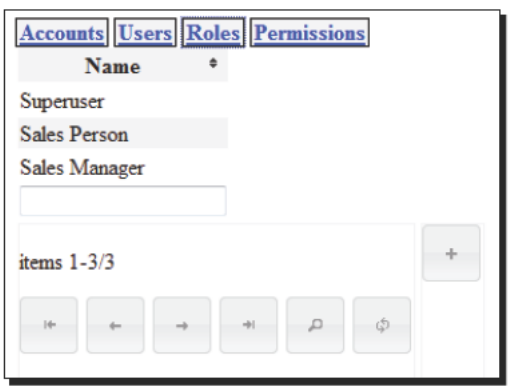
The aforementioned concepts described can be captured in this data model:



In our simplified model, a user can have one role, but a role can have one or more permissions. A permission consists of several attributes, an entity, action, and level, that together describe under which conditions something is allowed.

Time for action – implementing role-based access control

Run the example application provided in `access2.py` and log in as `admin`. You will see that besides **Users** and **Accounts**, you are presented with links to **Roles** and **Permissions** as well. If you click on **Roles**, you will see we have defined several roles:



As you can see in the screenshot, we have also defined a **Superuser** role to illustrate that it is possible to extend the concept of role-based access control to the maintenance of roles and permissions themselves.

What just happened?

Applications that use this form of access control have to be adapted only slightly. Take a look at `access2.py`:

Chapter9/access2.py

```
import os
import cherrypy

from rbacentity import AbstractEntity, Attribute, Picklist,
AbstractRelation

from browse import Browse
from display import Display

from logondb import LogonDB

db="/tmp/access2.db"
```

Compared to our previous example, only the first part is different, in that it includes the `rbacentity` instead of the `entity` module. This module provides the same functionality as the `entity` module, but the `AbstractEntity` class defined in this module has some added magic to provide access to roles and permissions. We will not inspect that in detail here, but will comment on it when we encounter it in the following code.

The next part is the definition of the `Entity` class. We could have opted for redefining the `AbstractEntity` class, but here we have chosen to add the functionality to the `Entity` subclass by adding and overriding methods where necessary:

Chapter9/access2.py

```
class Entity(AbstractEntity):
  database = db

  userentity = None
  logon = None

  @classmethod
  def setUserEntity(cls,entity):
    cls.userentity = entity

  @classmethod
  def getUserEntity(cls):
    return cls.userentity

  @classmethod
  def setLogon(cls,logon):
    cls.logon = logon

  @classmethod
  def getAuthenticatedUsername(cls):
```

```
    if cls.logon :
      return cls.logon.checkauth()
    return None

  def isallowed(self,operation):
    user = self.getUserEntity().list(
      pattern=[('name',self.getAuthenticatedUsername())])[0]
    entity = self.__class__.__name__
    if user.name == 'admin' :
      return True
    roles = user.getRole()
    if len(roles):
      role = roles[0]
      permissions = role.getPermission()
      for p in permissions :
        if p.entity == entity:
          if p.operation=='*' or p.operation==operation:
            if p.level == 0 :
              return True
            elif p.level == 1:
              for owner in self.getUser():
                if user.id == owner.id :
                  return True
    return False

  def update(self,**kw):
    if self.isallowed('update'):
      super().update(**kw)
```

Instead of just defining a `database` class variable, we now also define a `userentity` class variable to hold a reference to the class of the entity that represents a user and a `logon` class variable to hold a reference to a logon instance that can provide us with the name of an authenticated user.

This distinction is identical to examples in the previous chapters: we have a `User` entity in our main database where we may store all sorts of information related to the user (like full name, telephone number, gender, and so on) and a separate password database that holds just usernames and encrypted passwords. If the user is correctly authenticated against the password database, we know his/her username, which we can then use to retrieve the corresponding `User` instance with all the extra associated information. The class methods provide the means to get access to these class variables.

In this example, we only override the `update()` method (highlighted) but in a full implementation you might want to override other `Entity` methods as well. The pattern is simple: we call the `isallowed()` method with an argument that indicates which action we would like to check and if `isallowed()` returns `True`, we call the original method.

The first thing the `isallowed()` method itself does, is retrieve the username of the authenticated user with the `getAuthenticatedUsername()` class method. It then uses this name to find a `User` instance. Even though we might want to implement a role-based access scheme in our application to allow for the administration of roles and permissions by various users, we still provide a shortcut for the administrator here as a convenience (highlighted). This way we do not have to prime the database with roles and permissions for the admin user. For a real world application, you may choose differently of course.

Next we check if there are any roles associated with the user, and if this is the case, we retrieve all permissions associated with the first role (in this example, users have just one role). We then loop over all those permissions to check if there is one that applies to the entity we are interested in. If so, we check the `operation` field. If this field contains an asterisk (*) or is equal to the operation we are checking, we look at the `level`. If this `level` is zero, this means that the current user may perform this operation on this entity even if he/she is not the owner. If the level is one, he/she is only allowed to perform the operation if he/she owns the entity, so we check whether the user is in the list of users associated with the current entity.

Retrieving roles and permissions each time an operation is performed might incur a serious performance hit. It might be a good idea to cache some of this information. You have to be careful though and invalidate that cache as soon as the set of permissions for a user changes.

The next part of `access2.py`, as shown, illustrates how we may use this augmented version of the `Entity` class:

Chapter9/access2.py

```
class Relation(AbstractRelation):
    database = db

class User(Entity):
    name = Attribute(notnull=True, unique=True, displayname="Name",
primary=True)

class Account(Entity):
    name = Attribute(notnull=True, displayname="Name", primary=True)

class OwnerShip(Relation):
    a = User
    b = Account
```

```
class UserRoles(Relation):
    a = User
    b = User._rbac().getRole()
    relation_type = "N:1"

logon = LogonDB()

Entity.setUserEntity(User)
Entity.setLogon(logon)
```

As before, we define `User` and `Account` entities, and an ownership relation between them. The `rbacentity` module will have provided for `Role` and `Permission` classes and we can gain access to those with the `_rbac()` class method available to all `AbstractEntity` derived classes. The object returned by this `_rbac()` method provides a `getRole()` method that returns the class of the `Role` entity. We use it here to create a relation between users and their roles (highlighted). The final lines associate the password database and the `User` class with our new `Entity` class.

To provide access to the lists of roles and permissions, we can use the same `_rbac()` method to provide the `Role` and `Permission` classes needed to create `Browse` classes:

Chapter9/access2.py

```
class RoleBrowser(Browse):
    edit = Display(User._rbac().getRole(), edit=True, logon=logon)
    add = Display(User._rbac().getRole(), add=True, logon=logon)

class PermissionBrowser(Browse):
    edit = Display(User._rbac().getPermission(), edit=True,
 logon=logon, columns=User._rbac().getPermission().columns  +
[User._rbac().getRole()])
    add = Display(User._rbac().getPermission(), add=True,
logon=logon, columns=User._rbac().getPermission().columns + [User._
rbac().getRole()])
```

Summary

In this chapter, we filled in some gaps in our framework, Specifically, we learned how to implement more complex relations, for example, one-to-many and many-to-one relationships, how to create the necessary user interface components to maintain those relations, and how to implement role-based access control.

We're not quite there yet, because we are missing facilities to let end-users customize the datamodel, which is the subject of the next chapter.

10
Customizing the CRM Application

In this final chapter, we will add functionality to our framework to allow for some finishing touches.

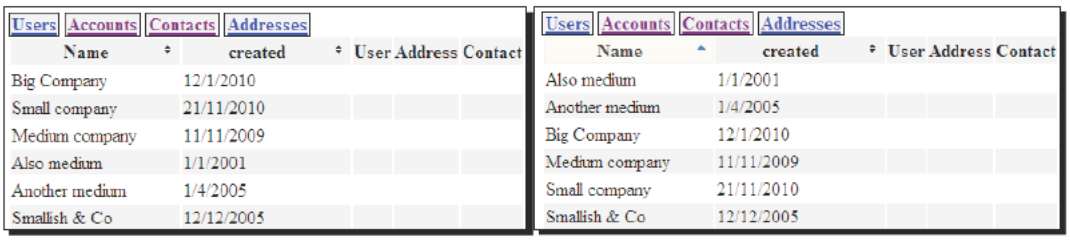
Specifically, we will see:

- What is needed to enhance the user interface to use the sort and filter capabilities in the framework
- How we can provide the end user with the means to customize an application without the need to program
- How to use these customizations to enhance the display of items and list of items
- How to enhance the stored information with information from external sources such as Google Maps

Time for action – sorting

When we implemented the `Browse` class in *Chapter 8*, *Managing Customer Relations*, together with the underlying functionality in the `listids()` method of the `AbstractEntity` class, we already took care of sorting and filtering.

We did not yet allow for any user interaction to make it actually possible to sort the list of entities shown. What was missing was the JavaScript code and some CSS to make this happen. Take a look at the following screenshot and notice the small arrow icons next to some headers on top of the list of accounts:



Users | Accounts | Contacts | Addresses

Name	created	User	Address	Contact
Big Company	12/1/2010			
Small company	21/11/2010			
Medium company	11/11/2009			
Also medium	1/1/2001			
Another medium	1/4/2005			
Smallish & Co	12/12/2005			

Users | Accounts | Contacts | Addresses

Name	created	User	Address	Contact
Also medium	1/1/2001			
Another medium	1/4/2005			
Big Company	12/1/2010			
Medium company	11/11/2009			
Small company	21/11/2010			
Smallish & Co	12/12/2005			

You can try the sort options for yourself when you run `crm2.py`.

Clicking once on a column marked with the small double arrows will sort the list on that specific column in ascending order. The header will change its background color to indicate that the list is now sorted and the small icon will change into a single up arrow.

Clicking it again will sort the list in descending order and this will be indicated by a small icon of a downward pointing arrow. A final click will render the list of items unsorted, as clicking the reset button will (not shown).

What just happened?

This sorting behavior is implemented by a few small parts:

- jQuery click handlers associated with the table headers
- Some CSS to style those headers with suitable icons
- Minor changes to the Python code that produces the table to make it simpler to track the sorting state in the browser.

First, let's see what has to be added to the JavaScript (the complete file is available as `browse.js`):

Chapter10/browse.js

```
$(".notsorted").live('click',function(){
    $("input[name=sortorder]").remove();
    $(".content form").first()
      .append('<input type="hidden" name="sortorder" value="'
              +$("div.colname",this).text()+',asc">');
    $("button[name=first]").click();
}).live('mouseenter mouseleave',function(){
    $(this).toggleClass("ui-state-highlight");
```

```
});
$(".sorted-asc").live('click',function(){
    //alert('sorted-asc '+$(this).text())
    $("input[name=sortorder]").remove();
    $(".content form").first()
      .append('<input type="hidden" name="sortorder" value="'
              +$("div.colname",this).text()+',desc">');
    $("button[name=first]").click();
}).live('mouseenter mouseleave',function(){
    $(this).toggleClass("ui-state-highlight");
});
$(".sorted-desc").live('click',function(){
    //alert('sorted-desc '+$(this).text())
    $("button[name=clear]").click();
}).live('mouseenter mouseleave',function(){
    $(this).toggleClass("ui-state-highlight");
});
```

Installing the click handlers is straightforward in itself, but what they have to accomplish is a little complicated.

The click handler must first determine which column will be used as the sort key. The element that is clicked on is available to the handler as `this` and this will give us access to a `<div>` element within the header that contains the column's name. This `<div>` element is not shown because its `display` attribute will be set to `none`. It is added because we need access to the column's canonical name. The `<th>` element itself contains just the display name of the column, which may be different from its canonical name.

This sort key will have to be passed to the application server and to this end we will use the machinery already in place: if we trigger submission of the form with the navigation buttons and make sure the proper sort parameters are passed along, we're almost there. How can this be accomplished?

jQuery provides convenient methods to insert new HTML elements into the existing markup (highlighted). From the name of the column, we construct a suitable value by appending either `asc` or `desc` to it, separated by a comma and use this as the value of a new hidden input element with a name of `sortorder` and insert this into the first `<form>` element with the `append()` method. The first form in the page is the form element containing the navigation buttons.

Because these same types of hidden `<input>` elements are used to maintain state when the user pages through the list of items, we first remove any `<input>` elements with a `name` attribute equal to `sortorder` to make sure these elements reflect the newly selected sort order and not any old one. The removal is accomplished by the aptly named `remove()` method.

The final step is to submit the form. We could trigger the submit event itself but because we have several buttons with a `type` attribute equal to `submit`, we have to be more specific.

It is not possible to trigger a `submit` event on a button, only on a form, but it is possible to trigger the `click` event on a button, thus mimicking the user interaction. Once the `click` event is triggered on the button with the `name` attribute of `first`, the form is submitted together with all its `<input>` elements, even hidden ones, including the new or replaced ones that indicate the sort order.

The handler for a `<th>` element that is already sorted in ascending order and marked by a `sorted-asc` class is almost identical. The only change we make is that the value of the hidden `<input>` element with `name=sortorder` is the column name with a `,desc` suffix instead of an `,asc` suffix.

Clicking on the `<th>` element when it is already sorted in descending order will cycle back to showing the unsorted state, so this click handler is even simpler as it just triggers the click handler of the clear button, which will result in an unsorted list.

The changes in the `index()` method in the `Browse` class are as follows (full code available as `browse.py`):

Chapter10/browse.py

```
yield '<thead><tr>'
          for col in self.columns:
                if type(col) == str :
                      sortclass="notsorted"
                      iconclass="ui-icon ui-icon-triangle-2-n-s"
                      for s in sortorder:
                            if s[0]==col :
                                  sortclass='sorted-'+s[1]
                                  iconclass=' ui-icon ui-icon-
triangle-1-%s'%({'asc':'n','desc':'s'}[s[1]])
                                  break
                      yield '<th class="%s"><div class="colname"
style="display:none">%s</div>'%(sortclass,col)+self.entity.
displaynames[col]+'<span class="%s"><span></th>'%iconclass
                else :
                      yield '<th>'+col.__name__+'</th>'
          yield '</tr></thead>\n<tbody>\n'
```

The Python code in our application barely has to change to accommodate this way of interaction. We merely adorn the `<th>` element of a column with a class that indicates the state of sorting.

It's either `notsorted`, `sorted-asc`, or `sorted-desc`. We also insert a `<div>` element to hold the true name of the column and an empty `<span>` element flagged with suitable jQuery UI icon classes to hold the icons that signal the sorting state (highlighted).

The `sortorder` list holds a number of tuples, each with the name of the column to sort as the first element and either `asc` or `desc` as the second element. This second element is used as the index into a dictionary that maps `asc` to `n` and `desc` to `s`, resulting in choosing either a `ui-icon-triangle-1-n` or a `ui-icon-triangle-1-s` class. Appending these classes together with a `ui-icon` class is all we need to let the jQuery UI stylesheets render our `<span>` element with a meaningful icon.

Many arrow-like icons, available in jQuery UI, follow a naming pattern similar to those for the small triangles here. The final part indicates a compass direction (here `n` for north, or upward) and the number indicates how many arrowheads are depicted in the icon (here just one, but there are many double-headed variants).

The resulting HTML for a column named `time` that is currently sorted in ascending order, would look something like this:

```
<th class="sorted-asc">
<div class="colname" style="display:none">time</div>
Time
<span class="ui-icon ui-icon-triangle-1-n"><span>
</th>
```

Besides the icon, we add some additional styles to `base.css` to make the headers more visible:

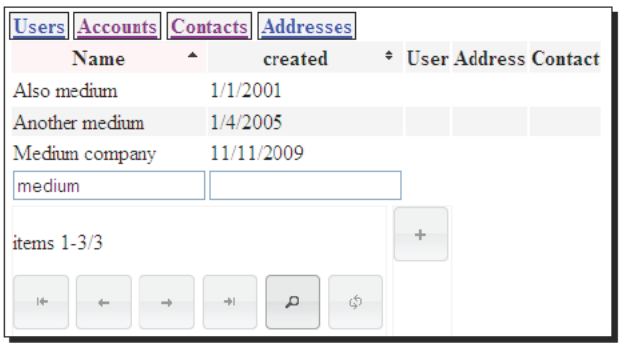
Chapter10/base.css

```
th.notsorted { padding-right:1px; border:solid 1px #f0f0f0; }
th.sorted-asc { padding-right:1px; border:solid 1px #f0f0f0;
background-color: #fff0f0; }
th.sorted-desc { padding-right:1px; border:solid 1px #f0f0f0;
background-color: #fffff0; }
th span { float:right; }
```

The table headers themselves are merely styled with a light gray color, but floating the `<span>` element that will hold the icon to the right is important, otherwise it would move below the text in the column header instead of the side.

Time for action – filtering

Almost the same approach we used for sorting can be used for filtering as well, only this time it is not a single click on a column header that does the trick, but we must provide the user with a way to enter the filter values. Take a look at the following screenshot or filter the data yourself by running `crm2.py` again:



If you insert values in any of the input fields below the columns in the table and click on the filter button (the one with the magnifying glass icon), the list of items to show is reduced to those items that match the filter values. Note that sorting and filtering may be combined and that clicking the clear button will remove both sorting and filtering settings.

What just happened?

Let's have a look at the JavaScript code:

Chapter10/browse.js

```
    $("button[name=search]").button({
      icons: {
          primary: "ui-icon-search"
          },
          text: false
}).click(function(){
          $("input[name=pattern]",
               $(".content form").first()).remove();
          $("input[name=pattern]").each(function(i,e){
                 var val=$(e).val();
                 var col=$(e).next().text();
                 $(".content form").first()
                  .append(
                  '<input type="hidden" name="pattern" value="'
                  +col+','+val+'">');
          });
```

```
            $("button[name=first]").click();
    });
```

Most of the work is done in the click handler of the search button. When the search button is clicked, we have to construct hidden `<input>` elements in the first form with a `name` attribute equal to `pattern` because it is those hidden filter inputs that will be passed to the server as arguments to the action URL when we trigger a submit of the form.

Note the second argument to the jQuery function (`$`) that selects an `<input>` element (highlighted). Both the visible `<input>` elements provided for the user to enter pattern values and the hidden ones in the form containing the navigation buttons have the same `name` attribute (`pattern`). We do not want to remove the visible ones as they contain the pattern values we are interested in. Therefore, we restrict the selection to the context of the first form, which is passed as the second argument.

After this, we are left with just the visible `<input>` elements which we iterate over with the `.each()` method. We collect both, the value of the `<input>` element and the content of its next sibling, which will be a (hidden) `<span>` element containing the true name of the column to filter. Together, these are used to construct a new hidden `<input>` element that is appended to the form that will be submitted.

After the elements are inserted, we submit this form by triggering the click handler of the submit button with the `name` attribute equal to `first`.

Chapter10/browse.py

```
        yield '<tfoot><tr>'
        for col in self.columns:
            if type(col)==str:
                filtervalue=dict(pattern).get(col,'')
                yield '''<td><input name="pattern"
                        value="%s"><span
                        style="display:none">%s</span>
                        </td>'''%(filtervalue,col)
        yield '</tr></tfoot>\n'
```

The only changes needed in the Python part of our web application are the insertion of suitable `<input>` elements that are initialized with the current pattern values to give a visible feedback to the user. The resulting HTML for a column that is currently filtered on the value ABC might look like this:

```
<td>
<input name="pattern" value="ABC">
<span style="display:none">name</span>
</td>
```

Customization

No matter how well designed and elaborate your application is, there is always more functionality that the user wants from it. Of course, with a proper framework in place and your code well documented, changes should not be a big problem, but on the other hand, you wouldn't want to restart an application just because some minor customizations are needed.

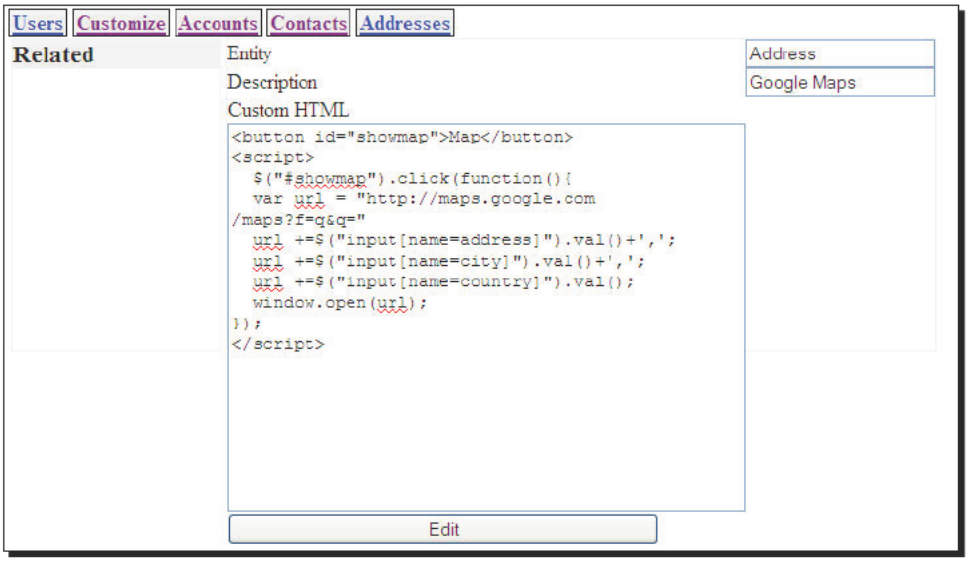
Many requests regarding an application will be concerned with the usability of the user interface, for example, different behavior of widgets or some additional features with regard to the information shown, like spell-checking entered text, finding stock market information related to a company shown, or the current exchange rate of a value on screen in a different currency. Quite a few companies including Microsoft, Yahoo!, and Google offer all sorts of free API's that may be used to enhance the display of values.

Time for action – customizing entity displays

Say we want to offer the end user the possibility to locate an address on Google Maps by simply clicking a button next to an address. Run `crmcustomize.py` and add a new address or edit an existing address. The edit/add screen will look similar to this:

When you click on the **Map** button, a new window will open, showing a map of that address as long as Google Maps was able to find it.

This functionality was added by the end user without the need to restart the server. Notice that in the opening screen, we have a new menu, **Customize**. If that menu is selected, we get a familiar looking interface showing a list of customizations added for different entities. If we double-click the one for `Address` with the Google Maps description, we get an edit screen, as shown in the following illustration:



A quick glance will show that the customization itself is simply HTML mixed with some JavaScript that is added to the markup produced by the application each time we open an edit or add screen for an `Address` entity.

It might not always be a good idea to allow any end user to customize an application. You might want to restrict some or all of the customization options to a subset of end users. Role-based access control is then again a suitable way to administer privileges.

What just happened?

Let's first have a look at the customization itself to get a feel of what can be accomplished. The code consists of a few lines of HTML and an embedded piece of JavaScript:

Chapter10/customization.html

```
<button id="showmap">Map</button>
<script>
$("#showmap").click(function(){
    var url = "http://maps.google.com/maps?f=q&q="
    url +=$("input[name=address]").val()+',';
    url +=$("input[name=city]").val()+',';
    url +=$("input[name=country]").val();
    window.open(url);
});
</script>
```

Because our application itself relies on jQuery, any customization code may use this library as well, so after we have defined a suitable button, we add a click handler to this button (highlighted) that constructs a Google Maps URL from the values of several `<input>` elements that will be present on the edit or add page of an `Address`, notably `address`, `city`, and `country`. This URL is then passed the `window.open()` method to open a new screen or tab with the results of this query.

Even better results may be obtained when the Google Maps API is used—see `http://code.google.com/intl/nl/apis/maps/documentation/javascript`.

What do we need to change in our framework to allow for this simple end user customization?

We need several related components to make this work:

- The `Display` class needs to be adapted to produce the custom code suitable for the instance that is shown.
- We need some way of storing the customization in the database together with the rest of the application.
- We need to allow a way to edit these customizations.

Let's look at these requirements in detail. The most important part is a way to store this information. Like we did for role-based access control, we can actually use our framework again; this time by defining a `custom` class. This `custom` class will create a `DisplayCustomization` class and provide access to it for all entities derived from the `AbstractEntity` class. The changes needed in the entity module are modest (the full code is available in `rbacentity.py`):

Chapter10/rbacentity.py

```
class custom:
    def __init__(self,db):
        class CustomEntity(AbstractEntity):
            database=db
        class DisplayCustomization(CustomEntity):
            entity = Attribute(notnull= True,
                displayname = "Entity")
            description = Attribute(displayname = "Description")
            customhtml = Attribute(displayname = "Custom HTML",
                htmlescape=True, displayclass="mb-textarea")
        self.DisplayCustomization = DisplayCustomization
    def getDisplayCustomization(self):
        return self.DisplayCustomization
```

```
    def getDisplayCustomHTML(self,entity):
          return "".join(dc.customhtml for dc in self.
DisplayCustomization.list(pattern=[('entity',entity)]))
```

Now that we have access to this storage for customization, any application can use it, but it also has to provide a way to let the application user edit these customizations. This entails defining a `Browse` class and adding a link to provide access to it. This is how it was done in the `crmcustomize` application, shown in the example (relevant changes only, full code available in `crmcustomize.py`):

Chapter10/crmcustomize.py

```
...
displaycustom = User._custom().getDisplayCustomization()

class DisplayCustomizationBrowser(Browse):
    edit = Display(displaycustom, edit=True, logon=logon)
    add = Display(displaycustom, add=True, logon=logon)

...
class Root():
    logon   = logon
    user    = UserBrowser(User)
    account = AccountBrowser(Account,
                columns=Account.columns+[User,Address,Contact])
    contact = ContactBrowser(Contact,
                columns=Contact.columns+[Address,Account])
    address = AddressBrowser(Address)
    displaycustomization = DisplayCustomizationBrowser(displaycustom,
                columns=['entity','description'])

    @cherrypy.expose
    def index(self):
          return Root.logon.index(returnpage='../entities')

    @cherrypy.expose
    def entities(self):
          username = self.logon.checkauth()
          if username is None : raise HTTPRedirect('.')
          user=User.list(pattern=[('name',username)])
          if len(user) < 1 : User(name=username)
          return basepage%'''<div class="navigation">
          <a href="user">Users</a>
          <a href="displaycustomization">Customize</a>
          <a href="account">Accounts</a>
          <a href="contact">Contacts</a>
          <a href="address">Addresses</a>
          </div><div class="content">
```

```
            </div>
            <script src="/browse.js" type="text/javascript"></script>
            '''
```

The final step is to enhance the display module with the means to retrieve and deliver these customizations. This is done by adding a few lines to the end of the `index()` method, as shown:

Chapter10/display.py

```
yield self.entity._custom().getDisplayCustomHTML('*')
yield self.entity._custom().getDisplayCustomHTML(self.entity.__name__)
```

Retrieving is straightforward enough and we actually retrieve two bits of customization: one for the specific entity we are showing and one for the customization code that is relevant for all entities. The user can add such customization with a special entity name of * (a single asterisk character). By putting the general customizations first in the markup we deliver, it is possible to override anything that is provided for the general case with customizations for the specific entities.

There is a bit of trickery needed elsewhere in the code of the `Display` class, however. Because the customization code may consist of HTML, including `<script>` elements containing JavaScript and `<style>` elements containing CSS, we might run into trouble when we display the forms to edit the customization code as these forms are HTML themselves. We, therefore, need some way to escape this code to prevent the content of the input box from being interpreted as HTML.

This is accomplished in the following way (the relevant changes to the `Attribute` class are shown):

Chapter10/rbacentity.py

```
class Attribute:
   def __init__(self,
          unique       =False,
          notnull      =False,
          default      =None,
          affinity     =None,
          validate     =None,
          displayname  =None,
          primary      =False,
          displayclass =None,
          htmlescape   =False):
          self.unique  =unique
          self.notnull =notnull
          self.default =default
```

```
self.affinity=affinity
self.coldef = (
 affinity+' ' if not affinity is None else '')
 + ('unique ' if unique else '')
 + ('not null ' if notnull else '')
 + ('default %s '%default if not default is None else '')
self.validate = validate?
self.displayname = displayname
self.primary = primary
self.displayclass = displayclass
self.htmlescape = htmlescape
```

The `Attribute` class provided in the entity module is extended to take an extra `htmlescape` parameter. If we set this to `True`, we indicate that the contents of this attribute should be escaped prior to showing it in a page.

The `MetaEntity` class will have to be extended as well to act upon these new features in the `Attribute` class:

Chapter10/rbacentity.py

```
classdict['htmlescape']={ k:v.htmlescape
          for k,v in classdict.items() if type(v) == Attribute}
```

The `MetaEntity` class is changed to store any `htmlescape` attributes in the `htmlescape` class attribute, a dictionary indexed by the attribute name.

At this point, we can create new entities with attributes marked for escape, but the `Display` class itself has to act on this information. We therefore add the following lines to the `index()` method of the `Display` class:

Chapter10/display.py

```
val=getattr(e,c)
if self.entity.htmlescape[c] :
       val=escape(val,{'"':'&quot;','\n':'&#xa;'})
       line='''<input type="text" name="%s"
              value="%s"
              class="%s">'''%(c,val,displayclass)
```

In the `index()` method of the `Display` class, before constructing an `<input>` element we can now check this `htmlescape` dictionary to see if we should escape the value of the attribute, and if so, use the `escape()` function provided in Python's `xml.sax.saxutils` module to convert any characters that might interfere.

A note of caution:

Allowing people to customize an application with HTML and JavaScript carries an inherent risk. When we developed a wiki application, we restricted the allowed input on pages by scrubbing the input of unwanted HTML. If you are serious about security (and you should be!), you have to think about what you will allow for customizations as well, especially to prevent cross-site scripting (XSS). Check, for example, `http://www.owasp.org/` for more on this and other security subjects.

Time for action – customizing entity lists

Of course, if we offer the user the opportunity to customize the display of *individual* entities, it makes sense to offer the same functionality for lists of entities. If you run `crm4.py` and click on the **Contacts** menu item, you will see a list as follows:

Users | Customize Item | Customize List | Accounts | Contacts | Addresses

First Name	Last Name	Gender	Telephone	Address	Account
Jane	Doe	Female	+3412345678		A Company
Jonathan	Doe	Male	+3412345678	Home address	A Company

You will notice that in the column containing the telephone numbers, those beginning with a plus sign are shown in a bold font. This will give a visible hint that this is probably a foreign number that needs some special code on your telephone switch.

What just happened?

The customization itself is a small piece of JavaScript that is inserted at the end of the page that shows the list of contacts:

Chapter10/customizationexample3.html

```
<script>
var re = new RegExp("^\\s*\\+");
$("td:nth-child(4)").each(function(i){
    if($(this).text().match(re)){
            $(this).css({'font-weight':'bold'})
    };
});
</script>
```

It uses jQuery to iterate over all `<td>` elements, which is the fourth child of their parent (a `<tr>` element, code highlighted). If the text contained in that element matches something that begins with optional whitespace and a plus sign (the regular expression itself is defined in the first line of the script), we set the `font-weight` CSS attribute of that element to `bold`.

Just as with the customization of `Display`, we need to add some way to store the customizations. The alterations to the `custom` class in the entity module are straightforward and copy the pattern set for `Display` (the complete code is available in `rbacentity.py`):

Chapter10/rbacentity.py

```
def __init__(self):
      ...
      class BrowseCustomization(CustomEntity):
            entity = Attribute(notnull= True,
                        displayname = "Entity")
            description = Attribute(displayname = "Description")
            customhtml = Attribute(displayname = "Custom HTML",
                  htmlescape=True, displayclass="mb-textarea")
      self.BrowseCustomization = BrowseCustomization
      ...
def getBrowseCustomization(self):
      return self.BrowseCustomization
def getBrowseCustomHTML(self,entity):
      return "".join(dc.customhtml
       for dc in self.BrowseCustomization.list(
                        pattern=[('entity',entity)]))
```

The definition of the `Browse` class in `browse.py` needs to be extended as well to retrieve and deliver the customizations (shown are the relevant lines from `browse.py`):

Chapter10/browse.py

```
yield self.entity._custom().getBrowseCustomHTML('*')
yield self.entity._custom().getBrowseCustomHTML(self.entity.__name__)
```

And the final step is to provide the user with a link to edit the customizations. This is done in the main application (available as `crm4.py`) by adding these lines, again following the pattern set for the display customizations (the lines relevant for the browse customizations are highlighted):

Chapter10/crm4.py

```
displaycustom = User._custom().getDisplayCustomization()
browsecustom = User._custom().getBrowseCustomization()
class DisplayCustomizationBrowser(Browse):
   edit = Display(displaycustom, edit=True, logon=logon)
   add = Display(displaycustom, add=True, logon=logon)
class BrowseCustomizationBrowser(Browse):
```

```
    edit = Display(browsecustom, edit=True, logon=logon)
    add = Display(browsecustom, add=True, logon=logon)

with open('basepage.html') as f:
    basepage=f.read(-1)

class Root():
    logon   = logon
    user    = UserBrowser(User)
    account = AccountBrowser(Account,
                columns=Account.columns+[User,Address,Contact])
    contact = ContactBrowser(Contact,
                columns=Contact.columns+[Address,Account])
    address = AddressBrowser(Address)
    displaycustomization = DisplayCustomizationBrowser(
                displaycustom,columns=['entity','description'])
    browsecustomization = BrowseCustomizationBrowser(
                browsecustom,columns=['entity','description'])

    @cherrypy.expose
    def index(self):
          return Root.logon.index(returnpage='../entities')

    @cherrypy.expose
    def entities(self):
          username = self.logon.checkauth()
          if username is None : raise HTTPRedirect('.')
          user=User.list(pattern=[('name',username)])
          if len(user) < 1 : User(name=username)
          return basepage%'''<div class="navigation">
          <a href="user">Users</a>
          <a href="displaycustomization">Customize Item</a>
          <a href="browsecustomization">Customize List</a>
          <a href="account">Accounts</a>
          <a href="contact">Contacts</a>
          <a href="address">Addresses</a>
          </div><div class="content">
          </div>
          <script src="/browse.js" type="text/javascript"></script>
          '''
```

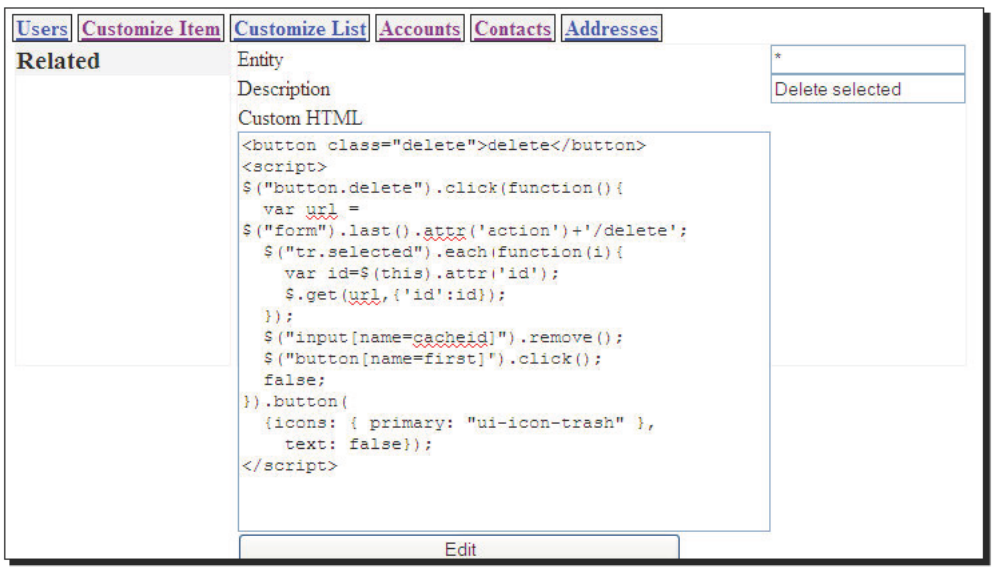
We are, of course, not restricted to actions on the client-side only. As we may utilize all the AJAX capabilities of jQuery, we can do quite elaborate things.

Our entity browser already has the functionality to mark a row as selected if we click it a single time. However, we did not implement any useful action to go with this selection.

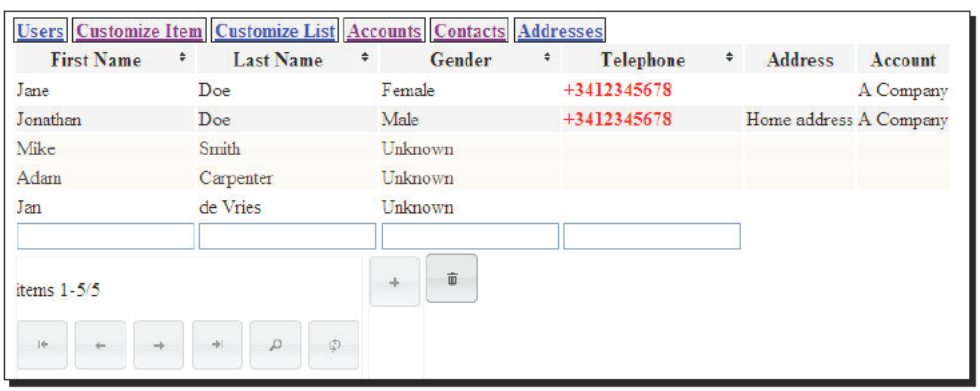
When we first implemented the `Display` class, we added a `delete()` method and exposed it to the CherryPy engine. We do not make use of this method in any way. Now that we can customize the entity browser, we can correct this and implement some functionality to add a button that when clicked will delete all selected entries. Mind you, it probably makes more sense to provide such basic functionality in a real application from the start, but it does show what is possible.

Time for action – adding a delete button

Run `crm4.py` once more, and in the **Customize List** menu, add an item that applies to all entities (and hence is marked as '*') as follows:



If we now open, for example, the list of contacts, we see a new button with a trashcan icon:



What just happened?

The customization we added consists of some HTML to define a `<button>` element and some JavaScript to render it as a nice trashcan button and to act on a click:

Chapter10/customizationexample4.html

```
<button class="delete">delete</button>
<script>
$("button.delete").click(function(){
    var url = $("form").last().attr('action')+'/delete';
    $("tr.selected").each(function(i){
          var id=$(this).attr('id');
          $.get(url,{'id':id});
    });
    $("input[name=cacheid]").remove();
    $("button[name=first]").click();
    false;
}).button({icons: { primary: "ui-icon-trash" },text: false});
</script>
```

The click handler fetches the `action` attribute from the last form in the list of entities (highlighted). This form holds the add button and this `action` attribute will therefore point to the URL serviced by the `index()` method of a `Display` instance. We simply add `delete` to it to make it point to the URL that will be serviced by the `delete()` method.

The next step is to iterate over all `<tr>` elements with a `selected` class and use jQuery's `get()` method to fetch the URL with the `id` attribute from the `<tr>` element added as an argument.

Finally, we have to redisplay the list of entities to show the effects of the deletion. If the list was filtered and/or sorted, we would like to retain that, but we still have to remove the hidden `<input>` element that holds the `cacheid`, otherwise we would be presenting the old list. After removing it, we trigger the click handler on the first button to initiate a reload.

Like almost every jQuery method, the `click()` method returns the selected elements it was invoked on, so we can chain a `button()` method to adorn our button element with a proper icon.

Summary

This final chapter was all about polishing up our CRM application. We enhanced the user interface to utilize the sort and filter features of the underlying framework, reused the framework itself to store and manage user customizations, and showed how powerful these customizations can be by enhancing the display of items and list of items by retrieving data from Google Maps.

A
References to Resources

Without repeating each and every reference given in the book, this appendix lists a number of resources that give good and comprehensive information on various subjects of interest to people building web applications.

Good old offline reference books

Sometimes it's nice to leave the keyboard and just relax with a book (or e-reader) and do some reading about our favorite subjects. The following small selection of books is reference work I pick up regularly (some ISBNs may reflect the e-book version):

Especially for people familiar with Python, having some good books around about JavaScript and the jQuery libraries is very convenient. The following three books are good starters:

- **Learning JavaScript, Second Edition, Shelley Powers, O'Reilly, 978-0-596-52187-5**

  Comprehensive introduction to JavaScript basics.

- **jQuery Cookbook, Cody Lindley, O'Reilly, 978-0-596-15977-1**

  A large selection of practical examples of how to solve common requirements with jQuery.

- **jQuery UI 1.7, Dan Wellman, Packt, 978-1-847199-72-0**

  A step-by-step explanation of all the functionality of the jQuery UI library, including advanced features like drag-and-drop.

Python has very good documentation online. Especially the coverage of the standard modules distributed with Python is excellent, but to get a thorough insight into the language itself and the features added in version 3, this book is a good start: **Programming in Python 3, Mark Summerfield, Addison Wesley, 978-0-32168056-3**

All of the following books cover Python subjects that play an important role in this book:

- **Python Testing Beginner's Guide, Daniel Arbuckle, Packt, 978-1847198-84-6**

  Testing doesn't have to be difficult and this book shows why.

- **CherryPy Essentials, Sylvain Hellegouarch, Packt, 978-1904811-84-8**

  The CherryPy application server that we use extensively in the examples in this book is capable of much more. Written by its primary developer, this book covers all the features and gives practical examples of some web applications as well.

- **Using SQLite, Jay A. Kreibich, O'Reilly, 978-0-596-52118-9**

  This book shows what SQLite is capable of and is even a good introduction to database design and the use of SQL. Not Python-specific (SQLite is used in many more places than Python alone).

- **Mastering Regular Expressions, Third Edition, O'Reilly, 978-0-596-52812-6**

  This book is everything there is to know about regular expressions. It's mostly not Python-specific, but as the regular expression library in Python closely resembles the one in Perl, almost all examples can be used as is in Python.

- **CSS Mastery, Andy Budd, Friends of Ed, 978-159059-614-2**

  Not all style issues are covered by jQuery UI of course and CSS can be tricky. This book is one of the most readable ones I've found.

Additional websites, wikis, and blogs

Additional information on the tools and resources used in the book can be found online.

Tools and frameworks

- `http://www.cherrypy.org/`

  The pure Python application server used in the examples in this book.

- `http://projects.apache.org/projects/http_server.html`

  Apache is much more than just a web server, but this links to this webserver workhorse directly.

- `http://jquery.com/`

  `http://jqueryui.com/`

  All about the JavaScript libraries used throughout this book to spice up the user interface.

- `http://www.prototypejs.org/`

  `http://www.sencha.com/products/extjs/`

`http://mootools.net/`

`http://dojotoolkit.org/`

Possible alternatives to the jQuery/jQuery UI libraries. Each has its own strengths and weaknesses.

- `http://sqlite.org/`

  `http://wiki.python.org/moin/DatabaseInterfaces`

  The home of the embedded database engine bundled with Python and a list of alternative database engines that work with Python.

- `http://www.aminus.net/dejavu`

  `http://www.djangoproject.com/`

  `http://www.sqlalchemy.org/`

  `http://elixir.ematia.de/trac/`

  Some quality object relational mappers.

- `http://subversion.apache.org/`

  `http://git-scm.com/`

  Both good! Widely used version management tools.

- `http://code.google.com/apis/libraries/devguide.html`

  `http://www.asp.net/ajaxlibrary/cdn.ashx`

  Content delivery frameworks may reduce the load on your own web server significantly.

- `http://pypi.python.org/pypi`

  The Python package index. Lists thousands of packages ready for use with Python. Check this first before reinventing the wheel.

- `http://www.datatables.net/`

  `http://www.appelsiini.net/projects/jeditable`

  Two very capable jQuery plugins. Both are excellent examples of how to extend jQuery.

- `http://getfirebug.com/`

  An extension for the Firefox browser. Invaluable when debugging web applications.

- `http://seleniumhq.org`

  A tool to test user interfaces/web pages.

- `http://www.owasp.org/index.php/Main_Page`

  Securing your application is very important. On this site, you will find information about general principles as well as specific attack patterns (and their remedies).

Newsfeeds

- `http://planet.python.org/`

  A large collection of blogs about Python.

- `http://michelanders.blogspot.com/`

  The author's blog about writing web applications in Python.

B
Pop Quiz Answers

Chapter 2, Creating a Simple Spreadsheet

Serving content with CherryPy

Answer:

Rename the `index()` method to `content()`

Remember that in order to serve the content referred to by a URL such as `http://127.0.0.1/content`, CherryPy looks for a method named `content()` in the object instance passed to the `quickstart()` function. Later on, we will see that it is also possible to build hierarchies of classes that enable CherryPy to serve URLs like `http://127.0.0.1/app/toplevel/content` as well.

Adding an icon to a button

Answer:

```
$("button").button({icons: {primary: 'ui-icon-refresh'}})
```

Like many jQuery and jQuery UI plugins, the button widget takes an `options` object as an argument. This `options` object may have a number of attributes, one of them—the `icons` attribute. The value of this attribute itself is an object again, its `primary` attribute determining which of the many standard icons will be displayed on the button. Refer to the online documentation of the button widget to see all options: `http://jqueryui.com/demos/button/` and check jQuery UI's themeroller page at `http://jqueryui.com/themeroller/` for an overview of all available icons for a given theme.

Adding conversions to a unitconverter instance

Answer:

```
$("#example").unitconverter({'cubic feet_litres':1.0/28.3168466 })
```

Changing option defaults

Answer: b

Chapter 3, Tasklist I: Persistence

Session IDs

Answer 1:

No, CherryPy will only save the session data to persistent storage if something is written to the session data while preparing a response. If an unknown session ID is received, the application cannot identify the user and will signal that to the client, but it will not store anything in the session data.

Answer 2:

c, because a client that doesn't store cookies will never send a request containing the session ID, the server will generate a new one.

Styling screen elements

Answer 1:

Either leave out the `text:false` in the `options` object passed to the `button()` function or explicitly show it with `text:true`.

Answer 2:

The `<div>` element that encloses the `<form>` element might be wider and an unsuitable background color may show up where the form isn't covering the full width.

Chapter 4, Tasklist II: Databases and AJAX

Using variable selection criteria

Answer:

```
cursor.execute('select * from task where user_id = ?',(username,))
```

A working implementation is available as `taskdb3.py`. Note that because there may be more than one placeholder present in a query, we pass the actual values for these placeholders as a tuple. A peculiarity of the Python syntax demands that a tuple is defined as parentheses containing a comma separated list of expressions and that a tuple consisting of a single item still has to contain a comma. `(username,)` is, therefore, a tuple with a single item.

Spotting the error

Answer:

```
test_number()
```

It will fail in the very first assertion with an output like the following:

```
python.exe test_factorial.py
.F.
======================================================================
FAIL: test_number (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_factorial.py", line 7, in test_number
    self.assertEqual(24,fac(4))
AssertionError: 24 != 12

----------------------------------------------------------------------
Ran 3 tests in 0.094s

FAILED (failures=1)
```

It still doesn't say what is wrong in the code, but now you know that the new implementation does not calculate the factorial of a number correctly. The solution might not be hard to spot this time: the `range()` function should be passed `2` as its first argument, because only 0 and 1 are treated as special-case in the code

Chapter 5, Entities and Relations

How to check a class

Answer:

Python's built-in function `issubclass()` can provide the information we need. Checking, for example, the `instance_a` attribute might be implemented like:

```
if not issubclass(instance_a, Entity) : raise TypeError()
```

How to select a limited number of books

Answer:

```
booksdb.list(offset=20,limit=10)
```

Chapter 6, Building a Wiki

Pop quiz

Answer:

The `id`, as it is unique as well.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z