

Quick answers to common problems

LaTeX Cookbook

Over 90 hands-on recipes for quickly preparing LaTeX documents to solve various challenging tasks

Stefan Kottwitz

[PACKT] open source* PUBLISHING
community experience distilled

LaTeX Cookbook

Over 90 hands-on recipes for quickly preparing LaTeX documents to solve various challenging tasks

Stefan Kottwitz

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

LaTeX Cookbook

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2015

Production reference: 1231015

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

ISBN 978-1-78439-514-8

www.packtpub.com

Credits

Author

Stefan Kottwitz

Reviewers

Brandon Amos

Jan Baier

Commissioning Editor

Usha Iyer

Acquisition Editor

Sonali Vernekar

Content Development Editor

Athira Laji

Technical Editor

Abhishek R. Kotian

Copy Editor

Brandt D'mello

Project Coordinator

Harshal Ved

Proofreader

Safis Editing

Indexer

Monica Ajmera Mehta

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

About the Author

Stefan Kottwitz studied mathematics in Jena and Hamburg. He worked as an IT administrator and communication officer onboard cruise ships for AIDA Cruises and for Hapag-Lloyd Cruises. Following 10 years of sailing around the world, he worked for the next few years specializing in network infrastructure and IT security. In 2014, he became a consultant for Lufthansa Systems. Currently, he works for Lufthansa Industry Solutions and designs and implements networks for new cruise ships.

In between contracts, he worked as a freelance programmer and typography designer. For many years, he has been providing LaTeX support on online forums. He maintains the web forums `LaTeX-Community.org` and `goLaTeX.de`, the TeX gallery site `TeXample.net`, the Q&A sites `TeXwelt.de` and `TeXnique.fr`, the `TeXdoc.net` service and the server for the UK TeX FAQ. He is a moderator of the TeX Stack Exchange site and on `matheplanet.com`.

He publishes ideas and news from the TeX world on his personal blog at `TeXblog.net`.

Before this book, he authored *LaTeX Beginner's Guide*, by *Packt Publishing* in 2011.

About the Reviewers

Brandon Amos is a PhD student at Carnegie Mellon University with research interests in machine learning and computer vision. He is a Linux and LaTeX enthusiast and actively releases research and hobby projects on the open source community at `http://github.com/bamos`.

Jan Baier is a young system engineer and administrator from the Czech Republic. He holds a master's degree in computer science from the Czech Technical University, Prague. In his spare time, he teaches Unix/Linux basics. He is a cofounder of amagical.net, a company that focuses on delivering information systems to small and medium companies. He tries to boldly go where no one has gone before, and he believes in the true power of open source.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface v

Chapter 1: The Variety of Document Types 1

Introduction 1
Writing a short text 4
Writing a thesis 8
Designing a book 16
Creating a presentation 24
Designing a curriculum vitae 33
Writing a letter 37
Producing a leaflet 40
Creating a large poster 45

Chapter 2: Tuning the Text 53

Introduction 53
Inputting accented characters 54
Improving justification and hyphenation 55
Converting numbers to words 57
Putting text into a colorful box 59
Visualizing the layout 62
Visualizing boxes of letters and symbols 66
Typesetting in a grid 67
Absolute positioning of text 71
Starting a paragraph with an over-sized letter 75
Fitting text to a shape 79
Creating a pull quote 82

Chapter 3: Adjusting Fonts 87
Introduction 87
Choosing a font for a document 90
Locally switching to a different font 94
Importing just a single symbol from a font family 97
Writing bold mathematical symbols 99
Getting the sans serif mathematics font 103
Writing double stroke letters as if on a blackboard 106
Enabling the searching and copying of ligatures 108
Suppressing ligatures 109
Adding a contour 110

Chapter 4: Working with Images 113
Introduction 113
Including images with optimal quality 114
Automating image positioning 116
Manipulating images 118
Adding a frame to an image 119
Cutting an image to get rounded corners 120
Shaping an image like a circle 122
Drawing over an image 124
Aligning images 126
Arranging images in a grid 128
Stacking images 130

Chapter 5: Beautiful Designs 133
Introduction 133
Adding a background image 133
Creating beautiful ornaments 138
Preparing pretty headings 142
Producing a calendar 144
Mimicking keys and menu items 147

Chapter 6: Designing Tables 149
Introduction 149
Creating a legible table 150
Adding footnotes to a table 154
Aligning numeric data 157
Coloring a table 160
Merging cells 162
Splitting a cell diagonally 166
Adding shape, shading, and transparency 168
Importing data from files 176

Chapter 7: Contents, Indexes, and Bibliographies 179
Introduction 179
Tuning a table of contents, lists of figures, and tables 180
Creating a bibliography 183
Adding a glossary 188
Creating a list of acronyms 191
Producing an index 193

Chapter 8: Getting the Most out of the PDF 197
Introduction 197
Adding hyperlinks 198
Adding metadata 203
Adding copyright information 206
Inserting comments 208
Producing fillable forms 211
Optimizing the output for e-book readers 215
Removing white margins 217
Combining PDF files 219
Creating an animation 220

Chapter 9: Creating Graphics 223
Introduction 223
Building smart diagrams 225
Constructing a flowchart 231
Building a tree 236
Building a bar chart 240
Drawing a pie chart 243
Drawing a Venn diagram 247
Putting thoughts in a mind map 250
Generating a timeline 253

Chapter 10: Advanced Mathematics 257
Introduction 257
Quick-start for beginners 258
Fine-tuning math formulas 266
Automatic line-breaking in equations 268
Highlighting in a formula 270
Stating definitions and theorems 274
Drawing commutative diagrams 281
Plotting functions in two dimensions 287
Plotting in three dimensions 292
Drawing geometry pictures 295
Doing calculations 303

Chapter 11: Science and Technology 307
Introduction 307
Typesetting an algorithm 308
Printing a code listing 311
Application in graph theory 315
Writing physical quantities with units 319
Writing chemical formulae 322
Drawing molecules 326
Representing atoms 332
Drawing electrical circuits 334
Chapter 12: Getting Support on the Internet 339
Introduction 339
Exploring online LaTeX resources 339
Using web forums 342
Framing a really good question 346
Creating a minimal example 346
Index 351

Preface

LaTeX is a high-quality typesetting software and is very popular, especially among scientists. Its programming language gives you full control over every aspect of your documents, no matter how complex they are. LaTeX's large number of customizable templates and supporting packages cover most aspects of writing with embedded typographic expertise.

With this book, you will learn to leverage the capabilities of the latest document classes and explore the functionalities of the newest packages.

This book starts with examples of common document types. It provides samples for tuning text design, using fonts, embedding images, and creating legible tables. Common parts of the document such as the bibliography, glossary, and index are covered with LaTeX's modern approach.

You will learn how to create excellent graphics directly within LaTeX; this includes creating diagrams and plots quickly and easily.

Finally, the book shows the application of LaTeX in various scientific fields.

The example-driven approach of this book is sure to increase your productivity.

What this book covers

Chapter 1, *The Variety of Document Types*, introduces you to the different types of documents and covers how LaTeX can be used for for any document type such as a thesis, a book, a CV, a presentation, a flyer, and a large poster.

Chapter 2, *Tuning the Text*, focuses on customizing text details in the documents. We will start with some very useful basics, covering some helpful things. We will end the chapter with recipes that show off what LaTeX can do beyond rectangular paragraphs.

Chapter 3, *Adjusting Fonts*, helps you to choose fonts globally, and demonstrates how to adjust them within the document.

Chapter 4, *Working with Images*, contains some recipes for including, positioning, and manipulating images within LaTeX. After discussing the quality aspects, you will get to know about the concept of floating figures for automated positioning to give well-balanced text heights on pages.

Chapter 5, *Beautiful Designs*, helps you in adding background images, creating beautiful ornaments, adding pretty headings, producing a calendar, and inserting symbols for computer keys and menu items.

Chapter 6, *Designing Tables*, helps you to create good-looking tables. Specifically, it covers creating a legible table, aligning numeric data, adding colors, shape, shading, and transparency merging and splitting cells, and reading in table data from external files.

Chapter 7, *Contents, Indexes, and Bibliographies*, provides recipes for quickly starting and customizing the table of contents, lists of of figures and tables, bibliographies, glossaries, and indexes.

Chapter 8, *Getting the Most out of the PDF*, explores the capabilities of PDF, such as metadata, PDF comments, and fillable forms. You will see how to combine PDF files, how to crop margins, and how to optimize the output for e-books.

Chapter 9, *Creating Graphics*, contains recipes to create impressive graphics. We will start off using modern packages for various purposes to create complete and useable graphics, such as various kinds of diagrams and charts.

Chapter 10, *Advanced Mathematics*, explores one of the classic strong points of LaTeX—its excellent quality in typesetting formulas. That's why LaTeX is the most used writing software in mathematics. When it comes to formulas, other sciences benefit from it as well.

Chapter 11, *Science and Technology*, deals with other sciences such as chemistry, physics, computer science, and technologies such as electronics. This chapter will be an overview, showing how LaTeX can be used in various fields using specific recipes.

Chapter 12, *Getting Support on the Internet*, starts with a guide to the most useful Internet resources for LaTeX. This chapter will show you how to get help from the TeX online communities.

What you need for this book

The software that you need here is TeX Live, version 2015 or later, or MiKTeX, version 2.9 or later. It is recommendable to also install a LaTeX editor. The introduction in Chapter 1 will explain, where to get the TeX software and editors. The Chapter 12 will guide you to further Internet resources.

An important tool is texdoc. It opens manuals and further documentation for you. texdoc is contained in TeX Live. MiKTeX provides a similar tool with the same name. texdoc is called at in Command Prompt by typing texdoc keyword. For packages and bundles, the keyword is usually just the name.

All the code examples in this book can be downloaded as described later in the section Customer support. So, you don't need to type the code, neither to do copy & paste. That's why we could split the codes into snippets to explain them step by step.

Who this book is for

If you already know the basics of LaTeX and if you'd like to get quick and efficient solutions to your problems, this is the perfect book for you. If you are an advanced reader, you can use this book's example-driven format to take your skillset to the next level. Some familiarity with the basic syntax of LaTeX and how to use the editor of your choice for compiling is required.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Write a mathematical equation using the `equation` environment."

A block of code is set as follows:

```
\begin{document}
\frenchspacing
\raggedbottom
\selectlanguage{american}
\pagenumbering{roman}
\pagestyle{plain}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **browse files** to have filler images and a sample file."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

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

Piracy

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

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

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

Questions

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

1
The Variety of Document Types

Documents can vary in size, format, sectioning, and design in general. You can use LaTeX for any document type. In this chapter, you will find recipes for different kinds of documents.

We will specifically cover the following topics:

- Writing a short text
- Writing a thesis
- Designing a book
- Creating a presentation
- Designing a curriculum vitae
- Writing a letter
- Producing a leaflet
- Creating a large poster

Introduction

LaTeX has been around for many years. Over time, developers and authors have contributed a lot of extensions to the LaTeX code base.

These extensions include the following:

- **Document class**: This is a style file which is the frame of your document. It usually comes with meaningful default settings that can be changed via options when loading it. Its macros can be customized by the `\renewcommand` command. It often provides commands for authors to modify settings.

- **Package**: This is a style file with a specific purpose that can be loaded in addition to the document class. Packages can be combined. Most of the time, we load many packages using the `\usepackage` command.
- **Bundle**: This is a set of closely related packages or classes. In our first recipe, *Writing a short text*, you will get to know some bundles.
- **Template**: This is a document filled with dummy text, which you can use it as a starting point, and just fill in your own headings, texts, formulas, and images. We will take a look at a template in our second recipe, *Writing a thesis*.

These add-ons are incredibly valuable! They are one of the reasons for the enduring success of LaTeX. We all agree that learning LaTeX can be hard with its steep learning curve. However, if you don't reinvent the wheel, and start with a good template or class and a quality set of packages, you can quickly achieve great results.

The purpose of this book is to assist you in this regard.

Getting ready

To be able to work with LaTeX, you need install the following on your computer:

- TeX and LaTeX software, which are together called **TeX distribution**
- A LaTeX editor, although you can use any text editor
- A PDF viewer to view the final output

If you already have these installed, great! In that case, you can skip the next paragraphs and immediately proceed to the first recipe.

A PDF reader is probably already installed on your computer, such as the Foxit Reader, the Adobe Reader or the Preview app on the Mac. Furthermore, most editors come with an integrated PDF previewer. So, let's take a look at the TeX software and editors.

TeX and LaTeX distributions

There are collections of TeX and LaTeX software that are ready to use and easy to install. Their websites provide information on installing and updating them. You may choose the download site for your system:

- **TeX Live**: At `http://tug.org/texlive/`, you can find download information and installation instructions for the cross-platform TeX distribution that runs on Windows, Linux, Mac OS X, and other Unix systems. It is supported by the TeX Users Group (TUG).
- **MacTeX**: This is based on TeX Live and has been significantly customized for Mac OS X. The basic information is available at `http://www.tug.org/mactex/`.
- **MiKTeX**: The download and documentation of that Windows-specific distribution can be found at `http://www.miktex.org/`.

- **proTeXt**: This is for Windows only, and it is derived from MiKTeX, but proTeXt is more user-friendly during installation. Its home page can be found at `http://www.tug.org/protext/`.

I recommend that you choose MacTeX if you have a Mac; otherwise, I recommend using TeX Live, since the support by the TUG is especially good.

Describing the setup is beyond the scope of this book. For TeX Live, you can find a step-by-step explanation with screenshots in the *LaTeX Beginner's Guide* by Packt Publishing. Generally, when you visit Internet addresses of the TeX distributions listed above, you can find detailed setup instructions.

Finally, on Linux, such as Ubuntu, Debian, Red Hat, Fedora, and SUSE versions, there's usually a TeX Live-based software package available via the operating system repositories. While it's usually not as up-to-date as an installation done via the TeX Live website or a TeX Live DVD, it's very easy to install using the Linux package manager, which you use to install any software usually.

LaTeX editors

There are many LaTeX editors, from small and quick to very feature-rich editors. The TeX distributions already provide the fine editor **TeXworks**, which I use. It can be set up together with TeX or with a package manager on Linux, and it can be downloaded from `http://www.tug.org/texworks/`.

I have a collection of the links to LaTeX editors and additional software on my blog, `http://texblog.net/latex-link-archive/distribution-editor-viewer/`, where you may look for alternative editors, which run on your operating system.

Additionally, there are pure online LaTeX editors, which run in a web browser, so you can use them even on tablets and smartphones. The most noticeable LaTeX editors are as follows:

- `https://www.overleaf.com`, with real-time collaborative editing and a rich text modes which renders headings, equations and further formatting directly in the editor
- `https://www.sharelatex.com`, also with real-time collaboration and revision history for tracking changes

If you need any help in setting up and using LaTeX or any other editor, you can visit a LaTeX web forum. In *Chapter 12*, *Getting Support on the Internet*, you can find links to these forums and see and how to use them. You can also find me on those forums.

If you would like to get help in learning LaTeX, you may take a look at *LaTeX Beginner's Guide* at `https://www.packtpub.com/hardware-and-creative/latex-beginners-guide`.

Once you have done the installation, you can launch the editor and start with a LaTeX recipe.

Writing a short text

While LaTeX is great for big documents, it's just as useful for smaller ones, and you get all the features to work with. Writing down homework or producing a seminar handout, for example, doesn't need book-like chapters, and the layout would not be very spacious. So we will choose a document class that suits the task at hand best.

There are class bundles that cover commonly used document types. Every LaTeX installation contains the base bundle with standard classes. There are class files for articles, books, reports, letters, and more. It is stable stuff; it has not really been changed for many years. If you don't care about the latest style, this can be sufficient. It would even run on a ten-year-old LaTeX installation.

In this recipe, we will use a class of the KOMA-Script bundle. This is a set of classes that were originally designed with the aim of replacing the standard classes and providing more features. In contrast to the rather static base bundle, KOMA-Script has been extensively developed during recent years. It became feature-rich and got an excellent user interface. Parts of its functionality are provided in packages that can be used together with other classes as well. You can identify KOMA-Script classes and packages by the prefix `scr`. This prefix stands for **script**, which was the initial name of this bundle.

How to do it...

We will start with a complete small document, already using various features. This can be your template, into which you can add your own text later on.

While we go through the document step by step, you may open the complete code directly with your editor, so you don't need to type it. It is contained in the code bundle available at the book's page `https://www.packtpub.com` and at the book's page `http://latex-cookbook.net`. Perform the following steps to create a small document:

1. Start with a document class. We will use the KOMA-Script class `scrartcl`, with A4 paper size, a base font size of 12 pt, and interparagraph spacing instead of the default paragraph indentation:

   ```
   \documentclass[paper=a4,oneside,fontsize=12pt,
     parskip=full]{scrartcl}
   ```

2. Begin the document:

   ```
   \begin{document}
   ```

3. Let LaTeX print a table of contents using this command:

   ```
   \tableofcontents
   ```

4. Start a section without numbering:

```
\addsec{Introduction}
```

5. Add some text:

```
This document will be our starting point for simple documents. It
is suitable for a single page or up to a couple of dozen pages.
The text will be divided into sections.
```

6. Start an automatically numbered section with text:

```
\section{The first section}
This first text will contain
```

7. Add a bulleted list using an `itemize` environment. Each list item starts with `\item`. Using `\ref{label}`, we will refer to labels, which we will create later:

```
\begin{itemize}
  \item a table of contents,
  \item a bulleted list,
  \item headings and some text and math in section,
  \item referencing such as to section \ref{sec:maths} and
        equation (\ref{eq:integral}).
\end{itemize}
```

8. Continue with text and start another numbered section:

```
We can use this document as a template for filling in our own
content.
\section{Some maths}
```

9. Set a label so that we can refer to this point whenever we want to refer to this section:

```
\label{sec:maths}
```

10. Continue with the text. We will start using some mathematical expressions in the text. We mark this by enclosing them in parentheses with a prefixing backslash, that is, `\( … \)`.

```
When we write a scientific or technical document, we
usually include math formulas. To get a brief glimpse
of the look of maths, we will look at an integral
approximation of a function \( f(x) \) as a sum with
weights \( w_i \):
```

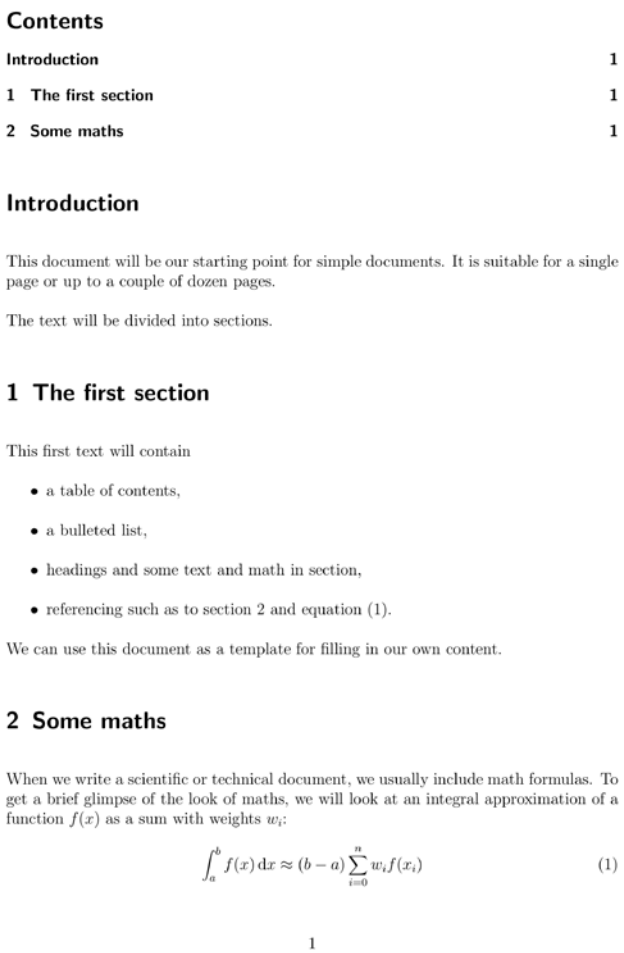
11. Write a mathematical equation using the `equation` environment. Again, place a label by using the following commands:

```
\begin{equation}
\label{eq:integral}
  \int_a^b f(x)\,\mathrm{d}x \approx (b-a)
  \sum_{i=0}^n w_i f(x_i)
\end{equation}
```

12. End the document:

```
\end{document}
```

13. Compile the document. The first page of the output will look like this:



Contents

Introduction 1

1 The first section 1

2 Some maths 1

Introduction

This document will be our starting point for simple documents. It is suitable for a single page or up to a couple of dozen pages.

The text will be divided into sections.

1 The first section

This first text will contain

- a table of contents,
- a bulleted list,
- headings and some text and math in section,
- referencing such as to section 2 and equation (1).

We can use this document as a template for filling in our own content.

2 Some maths

When we write a scientific or technical document, we usually include math formulas. To get a brief glimpse of the look of maths, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \qquad (1)$$

1

How it works...

In the first line, we loaded the document class `scrartcl`. In square brackets, we set options for specifying an A4 paper size with one-sided printing and a font size of 12 pt. Finally, we chose to have a full line between paragraphs in the output so that we can easily distinguish paragraphs.



The default setting is no space between paragraphs and a small indentation at the beginning of each paragraph. Uncheck the `parskip` option to see it. We chose a paragraph skip because many people are used to it when working with e-mails, while indentation costs line space, which is a precious resource on small electronic devices.

Without further ado, we began the text with a table of contents. While numbered sections are started by the `\section` command, we can start unnumbered sections by using the starred version `\section*`. However, we used the KOMA-Script command `\addsec` for the first unnumbered section. That's because, in contrast with `\section*`, the `\addsec` command generates an entry in the table of contents.

An empty line tells LaTeX to make a paragraph break.

As bulleted lists are a good way to clearly present points, we used an `itemize` environment for this. All environments start with the `\begin` command and are ended with the `\end` command.



If you would like to have a numbered list, use the `enumerate` environment.

An `equation` environment has been used to display a formula that is automatically numbered. We used the `\label` command to set an invisible anchor mark so we could refer to it using its label name by the `\ref` command and get the equation number in the output.



Choosing label identifiers:

It is a good practice to use prefixes to identify kinds of labels, such as `eq:name` for equations, `fig:name` for figures, `tab:name` for tables, and so on. Avoid special characters in names, such as accented alphabets.

Small formulas within text lines have been enclosed in `\( ... \)`, which provides inline math mode. Dollar symbols, `$ ... $`, can be used instead of `\( ... \)`, which makes typing easier. However, the use of parentheses makes it easier to understand where the math mode starts and where it ends, which may be beneficial when many math expressions are scattered throughout the text.

For further information on math typesetting, refer to *Chapter 10*, *Advanced Mathematics*, specifically to the recipe *Finetuning a formula*.

See also

The part of the document before the `\begin{document}` command is called the **preamble**. It contains global settings. By adding a few lines to our document preamble, we can improve our document and modify the general appearance. *Chapter 2*, *Tuning the Text*, starts with additions that are beneficial for small documents as well. They enable the direct input of accented characters and unicode symbols, and they improve justification and hyphenation.

In *Chapter 3*, *Adjusting Fonts*, you can find recipes for changing the fonts of either the whole document or of certain elements.

For further customization tasks, such as modifying page layout, and adding a title page, refer to the recipe *Designing a book* in the current chapter. We will take a look at such settings on the occasion of a book example.

Writing a thesis

When you plan to write a large document like a thesis, you have two choices: either choose a ready-made template, or set up your own document environment. If you have got limited time and need to start off with your thesis fast, a template can come to the rescue.

Beware of outdated and questionable templates found somewhere on the Internet. Look first at the date and at user opinions, such as on web forums. The age of a template is not a problem by itself, as LaTeX can run it the same way when it's been written. However, as LaTeX has developed, better solutions have come up over time. Legacy code may not benefit from it.

Some universities provide their own templates. That may be ok, because requirements would be met for sure; just check if it can be improved; for example, by replacing an obsolete package with its recommended successors.

A very good source for checking the quality of a template is the guide to obsolete commands and packages in **LaTeX2e**, which is also called **l2tabu**. You can open the English version by typing the `texdoc l2tabuen` command in Command Prompt, or by visiting `http://texdoc.net/pkg/l2tabuen`.

To be clear, the LaTeX base is stable and solid, but there are changes in community-contributed packages.

In the previous recipe, *Writing a short text*, we took the bottom-up approach and built a document from scratch, adding what we desired. Now we will go top-down; let's use and understand a complete template, removing what we don't need.

As we need to choose a template now, let's take a real gem. The `ClassicThesis` package by Prof. André Miede is a thesis template of very good quality. The design follows the classic guide to writing, *The Elements of Typographic Style* by Robert Bringhurst, we will see some particular points later in this recipe. Its implementation is thoughtful and modern. Originally written in 2006, it's maintained today as well, and shipped with TeX distributions.

Even if you would not use this specific template later, it shows how we can organize a large document.

Getting ready

Though the `ClassicThesis` package may already be installed in your TeX system, named `classicthesis.sty`, the whole template is an archive of files that should go into your working directory.

Download the `ClassicThesis` archive from CTAN by visiting `http://www.ctan.org/tex-archive/macros/latex/contrib/classicthesis/`.

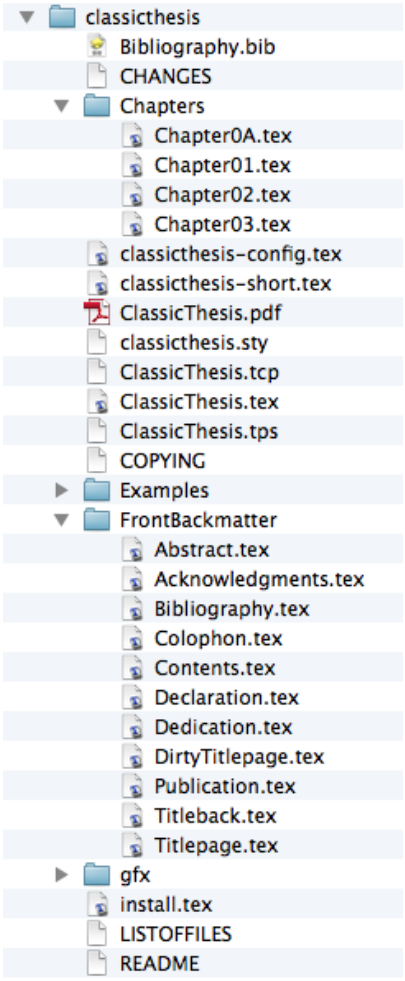
Instead of fetching single files, choose to download it as a `.zip` file. Unzip it to where you keep your personal documents, in its own directory. This directory will be your thesis directory.

This package provides a ready-made structured filesystem of the main document, style file, settings file, and document parts such as the abstract and foreword, and also chapters in dedicated files. You can edit all files and fill in your own text.

The `ClassicThesis.tex` file is the main document. Its filler text is the template's manual; this allows us to compile the template immediately for looking at the output design. By compiling, you can also verify that your TeX installation can handle additional packages if you need to install any.

How to do it...

After unzipping, your directory will have the following structure:



Now follow these steps:

1. Rename or copy the file `ClassicThesis.tex`; choose your own name, such as `MyThesis.tex`, but keep it in the same directory.
2. Open the main document, that is, `MyThesis.tex`, and look around to get a feeling of the document structure. Compile it for testing at least twice to get the correct referencing so you would know that this starting point works.

3. You can review and edit the settings in the main file, `MyThesis.tex`, and in the configuration file, `classicthesis-config.tex`. Over the course of the following pages, we will examine their content.
4. Open the existing `.tex` files, such as `Abstract.tex` and `Chapter01.tex`, with your editor. Remove the filler text and type in your own text. Add further chapter files as needed, and include them in the main file, `MyThesis.tex`, using the `\include` command. The structure is given; the technical part of the editing is like cloning files and copying lines, all you need to focus on is the actual thesis content now.

Don't worry if the font or margins don't please you yet. You can change the layout at any time. Let's take a closer look now, and then you will know how.

How it works...

We will take a look at the functional lines of the main file, `MyThesis.tex`.

The document preamble is as follows:

```
\documentclass[ twoside,openright,titlepage,
                numbers=noenddot,headinclude,
                footinclude=true,cleardoublepage=empty,
                BCOR=5mm,paper=a4,fontsize=11pt,
                ngerman,american,%
                ]{scrreprt}
\input{classicthesis-config}
```

The template is built on the KOMA-Script class, `scrreprt`. KOMA-Script as a LaTeX bundle is described in this chapter's first recipe, *Writing a short text*.

You can change the preset options to those you need, such as the font size or binding correction `BCOR`. There are many class options for adjusting the layout; you can read about them in the KOMA-Script manual. Type `texdoc scrguien` in Command prompt, or visit `http://texdoc.net/pkg/scrguien`. We will discuss some of them in depth in our next recipe, *Designing a book*.

Loading of packages and all the remaining settings is done in a single file, `classicthesis-config.tex`. We will look at it later in this recipe.

The document body starts with the following commands:

```
\begin{document}
\frenchspacing
\raggedbottom
\selectlanguage{american}
\pagenumbering{roman}
\pagestyle{plain}
```

The `\frenchspacing` line means that there's a single space following the punctuation after a sentence. Before the twentieth century, people stuffed extra space between sentences. LaTeX does this by default, or you can enter the `\nonfrenchspacing` line. The language is set to American English. Actually, you would need the `\selectlanguage` line only if you need to switch between languages for correct hyphenation.

We start with Roman page numbers. The `plain` page style means that we have no page headers for now, while page numbers are centered in the page footer.

Then, we look at the **front matter**, which is the part of the document where the formal parts before the actual content go:

```
\include{FrontBackmatter/DirtyTitlepage}
\include{FrontBackmatter/Titlepage}
\include{FrontBackmatter/Titleback}
\cleardoublepage\include{FrontBackmatter/Dedication}
%\cleardoublepage\include{FrontBackmatter/Foreword}
\cleardoublepage\include{FrontBackmatter/Abstract}
\cleardoublepage\include{FrontBackmatter/Publication}
\cleardoublepage\include{FrontBackmatter/Acknowledgments}
\pagestyle{scrheadings}
\cleardoublepage\include{FrontBackmatter/Contents}
```

Each commonly-required part of the front matter has its own file. Just edit the file as you need it, comment out by inserting a `%` symbol at the beginning of a line, or remove what you don't need. The `\cleardoublepage` line ends a page, but also ensures that the next page starts on the right-hand side. This can mean inserting an empty page if necessary; that is, a double page break. This will not happen if you change the `twoside` option to `oneside`, so you can keep the `\cleardoublepage` line, which will act like a `\clearpage` command when the `oneside` option is set.

Finally, we got the main matter:

```
\pagenumbering{arabic}
%\setcounter{page}{90}
% use \cleardoublepage here to avoid problems with pdfbookmark
\cleardoublepage
\part{Some Kind of Manual}
\include{Chapters/Chapter01}
\cleardoublepage
```

```
\ctparttext{You can put some informational part preamble text
here...}
\part{The Showcase}
\include{Chapters/Chapter02}
\include{Chapters/Chapter03}
```

In the main matter, we use Arabic page numbers. The `\pagenumbering` line resets the page number to `0`.

The thesis is divided into parts. Each one is split into chapters. You can omit the `\part` lines if you want your highest sectioning level to be the chapter level.

In the `Chapters` subdirectory, each chapter has its own `.tex` file; so, you can easily handle a large amount of text. Furthermore, you can use the `\includeonly` command to compile just the selected chapters for speeding up writing.

Finally, the main document ends with the **back matter**:

```
\appendix
\cleardoublepage
\part{Appendix}
\include{Chapters/Chapter0A}
%*******************************************************************
***
% Other Stuff in the Back
%*******************************************************
\cleardoublepage\include{FrontBackmatter/Bibliography}
\cleardoublepage\include{FrontBackmatter/Colophon}
\cleardoublepage\include{FrontBackmatter/Declaration}
\end{document}
```

The command `\appendix` resets the sectioning counters and changes them to alphabetic numbering, that is, the following chapters will be numbered A, B, and so on. Like in the front matter, the parts of the appendix are divided into several files.

Let's take a look at the configuration file. Open the `classicthesis-config.tex` file. The whole thing would take too much space in the book, so let's just take a look at some sample lines:

```
\newcommand{\myTitle}{A Classic Thesis Style\xspace}
\newcommand{\myName}{Andr\'e Miede\xspace}
\newcommand{\myUni}{Put data here\xspace}
\newcommand{\myLocation}{Darmstadt\xspace}
\newcommand{\myTime}{August 2012\xspace}
```

Here, you can fill in your own data. Besides being printed on the title page, this data will be used as metadata for the generated PDF document. There are more supported macros here, such as `\mySubtitle`, `\myProf`, and many more. The `\xspace` takes care of proper spacing after such a macro; that is, it inserts a space when there's no punctuation mark following it.

There's more...

As mentioned, this template contains design decisions inspired by the book *The Elements of Typographical Style* by Robert Bringhurst. The most notable design decisions are as follows:

- It doesn't use bold fonts, but rather uses small caps or italics elegantly to emphasize what's important.
- The text body is not very wide and this allows it to be read comfortably without needing the eyes to jump too far from the end of the line to the beginning of the next line. So, we have wide margins, which can be used for notes.
- The table of contents is not stretched to get right-aligned page numbers. To quote the author, "Is your reader interested in the page number or does she/he want to sum the numbers up?". That's why the page number follows the title.

Explore the `classicthesis-config.tex` file further to make modifications. Like in the previous recipe, we apply document-wide changes within the preamble, and here, we do it in this file.

We will now take a look at the selected lines of that configuration file.

Changing the input encoding

You might have noticed that special characters, such as accented characters, don't work directly in the input. Scroll to this line:

```
\PassOptionsToPackage{latin9}{inputenc}
\usepackage{inputenc}
```

This is another way of setting the input encoding. Refer to the recipe *Inputting accented characters* in *Chapter 2*, *Tuning the Text*, to understand the topic of encoding. Here, you should change the `latin9` option to `utf8` if you wish to use an UTF-8-capable system and editor. The `latin9` option is typically used by older editors on Windows. There are other encodings. If you would need another one, use `texdoc inputenc`, or visit `http://texdoc.net/pkg/inputenc` to take a look at the available options.

Getting a right-justified table of contents

The design is not set in stone, you may adjust it all. If you would like to see the page numbers in the table of contents right aligned, set a single option. Look at the very beginning of the `classicthesis-config.tex` file:

```
\PassOptionsToPackage{eulerchapternumbers,listings,drafting,%
  pdfspacing,%floatperchapter,%linedheaders,%
  subfig,beramono,eulermath,parts}{classicthesis}
```

Here, you can find the options for the actual `classicthesis` package. Simply add the `dottedtoc` option to this comma separated list. These options, among others, are documented in the template's manual. You can open it by typing `texdoc classicthesis` in Command Prompt, or at `http://texdoc.net/pkg/classicthesis`. (editor: please apply inline code formatting correctly, and URL formatting)

Changing the margins

To fulfill requirements of page margins or to implement your own layout ideas, you can specify exact page dimensions by loading the `geometry` package. Consider the following example:

```
\usepackage[inner=1.5cm,outer=3cm,top=2cm,bottom=2cm,
  bindingoffset=5mm]{geometry}
```

Here, you can also provide a value for the space that you leave for the binding and all margins you like. It's good practice to have a visible inner margin set to half the value of the outer margin, because margins will be added in the middle. For single-sided printing, with the `oneside` option, use `left=...` and `right=...` for the defining the margins instead of `inner` and `outer`.

Place such a line at the end of the `classicthesis-config.tex` file so that it will override the original settings.

Modifying the layout of captions

In the `classicthesis-config.tex` file, you can change the appearance of captions of figures and tables:

```
\usepackage{caption}
\captionsetup{format=hang,font=small}
```

The template loads the `caption` package. It provides a lot of features for fine-tuning captions. For now, the caption texts are indented so they will hang under the first line, and they will have a smaller font size than the normal text. By adding simple options, you can further adjust the appearance; for example, by adding the `labelfont=it` line, you can get italicized caption labels. Refer to the `caption` manual to learn more. As usual, you can open it via `texdoc caption` or at `http://texdoc.net/pkg/caption`.



While the `caption` package is a general solution working with most classes, including KOMA-Script, the KOMA-Script now offers extended integrated caption features.

Centering displayed equations

Another option is responsible for the alignment of displayed equations:

```
\PassOptionsToPackage{fleqn}{amsmath}
\usepackage{amsmath}
```

With these `ClassicThesis` lines displayed equations will be left aligned. The `fleqn` option allows switching to this alignment. If you would like to restore the default behavior, which is centering the equations, remove that first line that passes the option, or comment it out by a `%` symbol. But keep the second line, though, which loads the `amsmath` option, as this is the de-facto standard package for typesetting mathematics in LaTeX.

You can find further thesis templates at `http://www.latextemplates.com`.

See also

In the following chapters, you can find a lot of recipes for content elements in your thesis. For a beautiful thesis, elegant tables are of great value, so you may also want to take a look at the *Advanced table design* recipe in *Chapter 5, Beautiful Designs*.

Designing a book

A book can be a pretty large document, so we can take a similar approach to the one we took in the preceding recipe. Refer to that recipe to see how to split your document into handy files and how to organize the directory structure.

Commonly, books are printed two sided. They are divided into chapters, which start on right-hand side pages, and they have pretty spacy headings and often a page header showing the current chapter title. Readability and good typography are important, so you will hardly find books with an A4 paper size, double-line spacing, and similar specs, which some institutes expect of a thesis. This is why we have dedicated book classes with meaningful default settings and features.

How to do it...

As explained in the *Writing a short text* recipe, our choice will be a KOMA-Script class; only, this time it has the name `scrbook`.

Perform these steps to start off a book:

1. Start with the `scrbook` class and suitable options for paper and font size:

```
\documentclass[fontsize=11pt,paper=a5,
  pagesize=auto]{scrbook}
```

2. Specify the input encoding, which will depend on your editor settings:

```
\usepackage[utf8]{inputenc}
```

3. Choose a font encoding. T1 is good for European, English, or American texts:

```
\usepackage[T1]{fontenc}
```

4. If you want a nondefault font, load it. Here, we chose Latin Modern:

```
\usepackage{lmodern}
```

5. We will load the `blindtext` package to get English dummy texts. It also requires you to load `babel` with the English setting:

```
\usepackage[english]{babel}
\usepackage{blindtext}
```

6. Load the `microtype` package for better text justification:

```
\usepackage{microtype}
```

7. Using this command, you can switch off the additional space after sentence punctuation:

```
\frenchspacing
```

8. Begin the document:

```
\begin{document}
```

9. Provide a title, a subtitle, an author name, and a date. You can also set an empty value if you don't want to have something in the title field:

```
\title{The Book}
\subtitle{Some more to know}
\author{The Author}
\date{}
```

10. Let LaTeX print the title page:

```
\maketitle
```

11. Print out the table of contents:

```
\tableofcontents
```

12. We will divide this book in to parts, so start with the first part:

```
\part{First portion}
```

13. Start a chapter with a heading. It's good to have text before adding another heading, so let's insert some:

```
\chapter{The beginning}
Some introductional text comes here.
```

14. As we did in our first recipe, add a section and text, and another part with a chapter and section. Using the `\Blindtext` command, you can generate a long dummy text, and using `\blindtext`, you can generate shorter dummy texts. The `\appendix` command switches the font to alphabetic numbering:

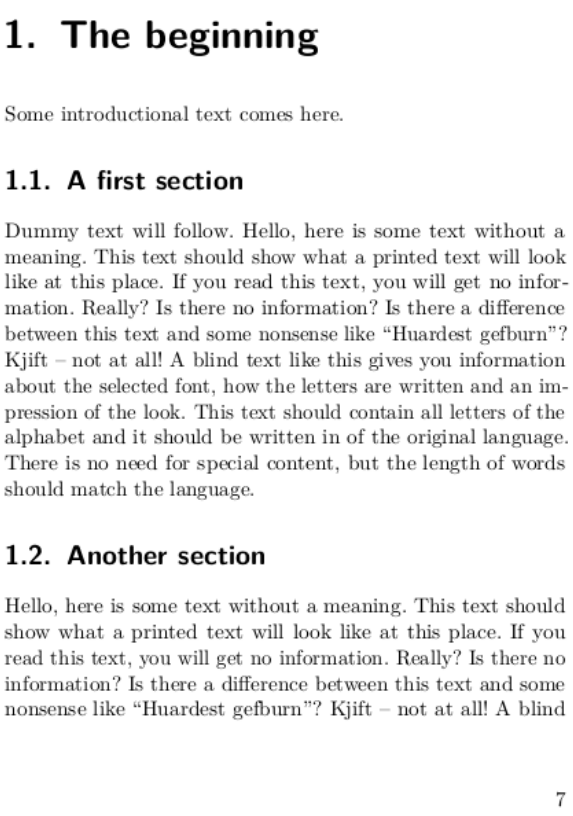
```
\section{A first section}
Dummy text will follow.
\blindtext
\section{Another section}
\Blindtext
\appendix
\part{Appendix}
\chapter{An addendum}
\section{Section within the appendix}
\blindtext
```

15. End the document:

```
\end{document}
```

16. Let your editor compile the text to PDF. You will get a 13-page-long book document with A5 paper size, a title page, part pages, chapter, section headings, and filler text.

Take a look at the following sample page:



1. The beginning

Some introductional text comes here.

1.1. A first section

Dummy text will follow. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.2. Another section

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind

7

Note the headings in sans serif font. This is an intentional default setting in KOMA-Script classes, which makes the headings lighter than the standard LaTeX big, bold, and serif headings. You may know the standard look.

17. Now, you can fill in your own text, add chapters and sections, and possibly add features, which will be described later in this recipe.

How it works...

At first, we loaded the class `scrbook`, which was specifically made for writing books. So, it is ready for two-sided printing with meaningful margins and good proportions of headings and text.

Besides the class's default settings, we chose a font size of 11 pt and A5 paper size, which is handy for a book. The `pagesize=auto` option is important here as it ensures that the A5 printing area will be taken over to the PDF page size.

Then, we did the following things, which will be explained in more detail at the beginning of *Chapter 2*, *Tuning the Text*:

- We declared the input character encoding and chose UTF-8, Unicode
- We set the font encoding to T1 by loading the `fontenc` package
- We chose the high-quality T1 supporting font set Latin Modern by loading the `lmodern` package
- We loaded the `babel` package with support for English
- We used the `microtype` package to get finer typography

The last package we loaded is the `blindtext`. You don't need it in your final document; here, it will serve us to provide filler text. Using such dummy text, we can get a better idea of the final result before writing the actual content.

Finally, we switched to the so-called French spacing, which means that after ending a sentence, we will get a normal interword space, not a wider space.

There's more...

You can change the layout of the book in many ways. Choose your settings at the beginning, or even better—start writing your content without any hesitation—once you have a decent amount of text, you can better see the effects of layout changes. You can do this at any time. Let's take a look at some design ideas.

Changing the page layout

When the book is bound after printing, the binding can cost space, that is, there may be less of the inner margin visible. You can specify a binding correction to compensate and to preserve layout proportions. So, if you would see 5 mm less of the inner margin after binding, add `BCOR=5mm` as class the option at the beginning. A similarly produced book may give you an idea of a good value.

The actual text area has the same ratios as the page itself. This is automatically done by a dividing construction, described in the KOMA-Script manual. It's really worth reading. You can open it by typing the `texdoc scrguien` command in Command Prompt, or online at `http://texdoc.net/pkg/scrguien`. The abbreviation, scrguien, comes from `scr` for the original package name Script, `gui` for guide, and `en` for English, and, obviously, from the ancient limit of eight characters per file name in older filesystems.

Besides the page and text area ratios, the result shows a bottom margin twice as high as the top margin, and an outer margin with double the width of the inner margin. Imagine an opened book, where the inner margins together appear with the same space as an outer margin. Sometimes, people make the mistake of thinking that the inner margin should be much bigger because of the binding, but that's done by raising the `BCOR`, as shown previously.

If you would like to get a bigger text area, which means narrower margins, you can still keep the ratios as described. Just raise the division factor of the aforementioned internal construction and check whether or not it would suit you. For example, set the class option as `DIV=10`; higher values are possible. That's a safe and easy way to preserve sane layout proportions.

To sum up, our example with 5 mm binding loss and pretty narrow margins can start like this:

```
\documentclass[fontsize=11pt,paper=a5,pagesize=auto,
  BCOR=5mm,DIV=12]{scrbook}
```

Alternatively, you can freely choose text and margin dimensions, when requirements of the publisher or institute need to be met. This can be done by loading the classic `geometry` package with the desired measurements, like we saw in the *Writing a thesis* recipe:

```
\usepackage[inner=1.5cm,outer=3cm,top=2cm,bottom=4cm,
  bindingoffset=5mm]{geometry}
```

Designing a title page

You can create your own title page, to present some more information in any style you desire. Let's take a look at an example that shows some handy commands for it.

Remove the `\maketitle` command. You can do the same thing with the commands `\title`, `\subtitle`, `\author`, and `\date`. Instead, put the `titlepage` environment right after the `\begin{document}` block:

```
\begin{titlepage}
  \vspace*{1cm}
  {\huge\raggedright The Book\par}
  \noindent\hrulefill\par
  {\LARGE\raggedleft The Author\par}
  \vfill
  {\Large\raggedleft Institute\par}
\end{titlepage}
```

The `titlepage` environment creates a page without a page number on it. We started with some vertical space using the `\vspace*` command. The `\vspace` command adds vertical space, which can be of a positive or negative value. Here, note the star at the end; `* \vspace` also works at the very beginning of a page, where a simple `\vspace` would be ignored. This default behavior prevents undesired vertical space at the top of a page, which originally may have been intended as space between texts.

We enclosed each line in curly braces. This is also called **grouping**, and it is used to keep the effect of changes, such as the font size, local within the braces. In each line, we do the following:

- Switch to a certain font size
- Choose left or right alignment
- Write out the text
- End with a paragraph break

The `\par` command is equivalent to an empty line in the input. Sometimes, people use it to keep the code compact, as we did here. We need to end the paragraph before the font size changes because that size defines the space between lines. Hence, we ended the paragraph before we closed the brace group. It's good to keep this in mind for when texts are longer.

Our only nontext design element is a modest horizontal line made by the `\hrulefill` line. The preceding `\noindent` line just prevents an undesired paragraph indentation, so the line really starts at the very left.

The `\vfill` line inserts stretching vertical space, so we get the last line pushed down to the title page bottom.

We took this occasion to show some commands for positioning text on a page. You can experiment with the `\vspace` and `\vfill` commands and their horizontal companions `\hspace` and `\hfill`. Just avoid using such commands to fix local placement issues in the actual document, when it would be better to adjust a class or package setting document wide. Use these only in the final stage to make tweaks if any are required.



The `titlepages` package provides 40 example title pages in various designs with full LaTeX source code. You can choose one, use it, and customize it.

Adding a cover page

The title page, which we produced previously, is actually an inner page. That's why it follows the normal page layout with the same inner and outer margin as the body text.

The cover is different, for example, it should have symmetric margins, and it can be designed according to your preference or individual choice. To get that deviating layout, it is recommended that you use a separate document for it. Another reason is that it will usually be printed on a different type of paper or cardboard.

So, you can start with an article-like class like the one we used in our first recipe, *Writing a short text*. Use options such as `twoside=false` or the equivalent `oneside` option to get symmetric margins. Then, you can proceed to positioning your text like we did with the title page.

Changing the document class

A very well designed book class is `memoir`. It is pretty complete in itself, so you don't need to load many packages as it already integrates a lot of features of other packages, providing similar interfaces. The `memoir` class has a monolithic, easy-to-use approach, but it needs to take care of package conflicts. It is not as flexible as choosing the package set by yourself. KOMA-Script, in contrast, provides its features mostly in packages that can also be used together with other classes. You can change to `memoir` this way:

1. Start off with the `memoir` class by changing the first line to the following:

   ```
   \documentclass[11pt,a5paper]{memoir}
   ```

2. Remove the unsupported `\subtitle` command.
3. To have the title on its own page, surround the `\maketitle` line with a `titlingpage` environment:

   ```
   \begin{titlingpage}
   \maketitle
   \end{titlingpage}
   ```

4. Compile, and compare.

The `memoir` class provides an extensive manual that can help you to customize your document. It's split in two parts. Type the `texdoc memman` command in Command Prompt to read the actual manual, and the `texdoc memdesign` command to read the part on book design, which is a great resource independent of the class. Alternatively, you can find these manuals at `http://texdoc.net/pkg/memman` and `http://texdoc.net/pkg/memdesign` respectively.

Another great start with a special beauty is the `tufte-latex` class. It comes with a `sample-book.tex` file, which you can also download from `http://ctan.org/tex-archive/macros/latex/contrib/tufte-latex`. You can open this book file, which contains some dummy content, and fill in your own text. One of its outstanding features is a wide margin for extensive use of side notes and small figures in the margin.

See also

A book may contain other elements, such as an index, a glossary, and a bibliography. Refer to *Chapter 7*, *Contents, Indexes, and Bibliographies*, which contains such recipes.

Creating a presentation

At a conference or at a seminar, speakers often use a projector or a screen for presenting written information in addition to what they are addressing. Such a presentation document requires a specific kind of layout and features.

In our recipe, we will use the `beamer` class, which has been designed specifically for this purpose. It provides the following features:

- A typical landscape slide format, here 128 mm x 96 mm
- Structured frames with dynamic information; for example, about sectioning
- Support for overlays and transition effects
- Predesigned themes for easily choosing the look
- A smart interface for customizing

How to do it...

We will start off with a sample presentation document, which we can extend. Follow the given steps:

1. Start with the `beamer` document class:

```
\documentclass{beamer}
```

2. Choose a theme. Here, we take the theme called `Warsaw`:

```
\usetheme{Warsaw}
```

3. Begin the document:

```
\begin{document}
```

4. Provide a title, subtitle, the names of the author and the institute, and a date:

```
\title{Talk on the Subject}
\subtitle{What this is about}
\author{Author Name}
\institute{University of X}
\date{June 24, 2015}
```

5. Make a slide using the `frame` environment. The first one will contain the title page:

```
\begin{frame}
  \titlepage
\end{frame}
```

6. Make a frame for the table of contents, with the title `Outline`. Add the option `pausesections`, so the table of contents will be shown stepwise, section by section:

```
\begin{frame}{Outline}
  \tableofcontents[pausesections]
\end{frame}
```

7. Start a section and a subsection within it:

```
\section{Introduction}
\subsection{A subsection}
```

8. All visible content goes into the `frame` environment, including lists, which are visually better than normal text in a presentation:

```
\begin{frame}{Very Informative Title}
  \begin{itemize}
    \item First thing to say.
    \item There is more.
    \item Another short point.
  \end{itemize}
\end{frame}
```

9. This frame will show an emphasized block with the title:

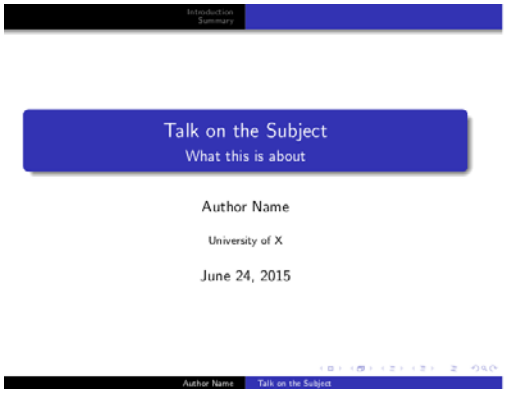
```
\begin{frame}{Another Title With Uppercased Words}
  Text
  \begin{alertblock}{A highlighted block}
    Some important information put into a block.
  \end{alertblock}
\end{frame}
```

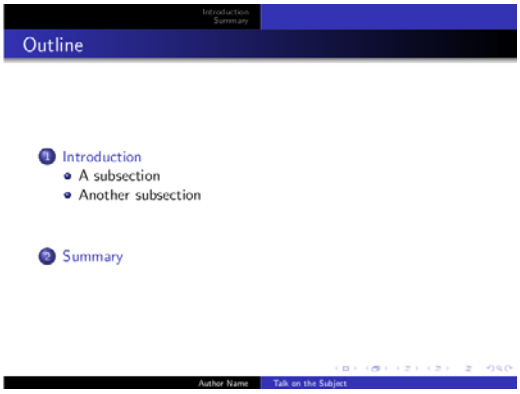
10. Add another subsection with a frame with a titled block. Also, add a new section with a slide containing a list. We have highlighted some words by using the `\alert` command. Finally, end the document:

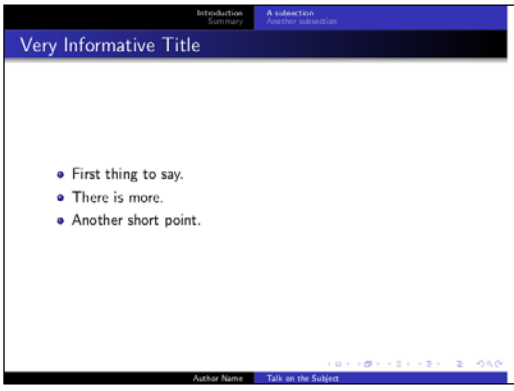
```
\subsection{Another subsection}
\begin{frame}{Informative Title}
  \begin{exampleblock}{An example}
    An example within a block.
  \end{exampleblock}
```

```
    Explanation follows.
  \end{frame}
  \section{Summary}
  \begin{frame}{Summary}
    \begin{itemize}
      \item Our \alert{main point}
      \item The \alert{second main point}
    \end{itemize}
    \vfill
    \begin{block}{Outlook}
      Further ideas here.
    \end{block}
  \end{frame}
  \end{document}
```

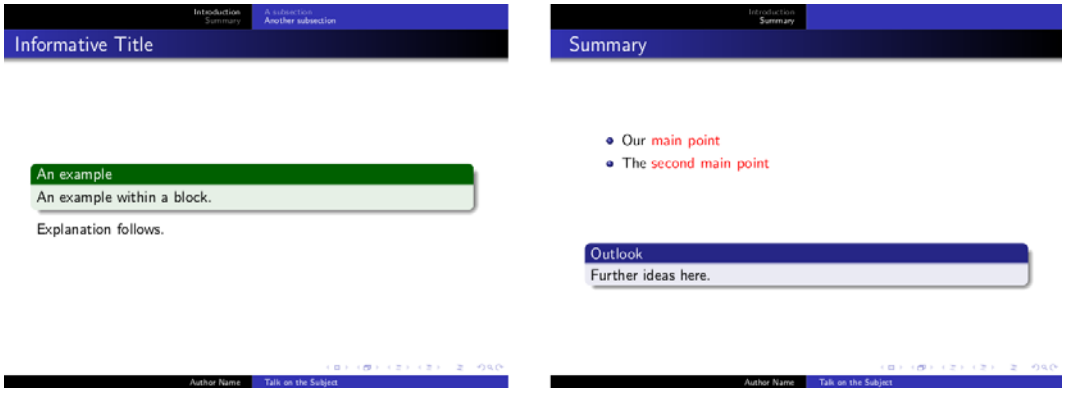
11. Compile, and take a look at the slides we have produced:











12. Adjust the title, name of the author, and date.
13. Edit the text in the frames. Add your own frames with titles and text. Use the components you found here, such as bulleted lists and boxes.

How it works...

We loaded the `beamer` class and chose the theme with the name `Warsaw`. You can easily replace it with another theme's name, compile it, and cycle through the themes until you find the best one for your occasion. By default, the following themes are installed:

- `AnnArbor`
- `Antibes`
- `Bergen`
- `Berkeley`
- `Berlin`
- `Boadilla`
- `boxes`
- `CambridgeUS`
- `Copenhagen`
- `Darmstadt`
- `default`
- `Dresden`
- `EastLansing`
- `Frankfurt`
- `Goettingen`
- `Hannover`

- Ilmenau
- JuanLesPins
- Luebeck
- Madrid
- Malmoe
- Marburg
- Montpellier
- PaloAlto
- Pittsburgh
- Rochester
- Singapore
- Szeged
- Warsaw

You can see these themes in the gallery at `http://latex-beamer.net`. We specified the title, subtitle, author, and date of the seminar, which is then printed by `\titlepage`. We used a `frame` environment, which we have to use for each slide.

The next frame contains the table of contents. We provided the frame title `Outline` as an argument to the frame in curly braces. For the `\tableofcontents` command, we added the `pausesections` option. As a result of doing this, section titles are printed one-by-one with a pause in between. This gives us the opportunity to explain what the audience will hear before they read further.

Here, we used the `\section` and `\subsection` commands just like in a normal LaTeX document. The heading is not directly printed. The sections and subsections are printed in the frame margin with the current position highlighted.

To get a bulleted list, we used an `itemize` environment just as we would in a normal LaTeX document. The environments `enumerate` for numbered lists and `description` for descriptive lists also work in the `beamer` frames.

To highlight information, we used the so-called `block` environments. Besides the standard `block` environment, we can use the `exampleblock` and `alertblock` commands to get a different style or color. The chosen theme determines the appearance of those blocks.

The `\alert` command is used to emphasize more distinctly, as seen in the last frame.

Now, you have the template and tools to create your presentation.



Consider the following while designing a presentation:

- Keep time constraints in mind; a frame per minute is a good rule of thumb.
- Use few sections, logically split in subsections; it is better to avoid subsubsections.
- Use self-explanatory titles for sectioning and frames.
- Bulleted lists help to keep things simple.
- Consider avoiding numbering references; one rarely cares about a reference to theorem 2.6 during a talk.
- Don't disrupt the reading flow with footnotes.
- Graphics, such as diagrams, help the audience with visualization.
- Slides should support your talk, not the other way round. Did you already bear with a presentation where the speaker just read aloud the text from the slides and used fancy transition effects? You can do it better.

There's more...

The `beamer` package has unique capabilities and an extraordinary design. We will explore it in the following pages.

Using short titles and names

Besides the title page, the title of the presentation and the author's name are additionally printed at the bottom of each frame. The exact position depends upon the chosen theme.

However, for long titles or names, the space might be insufficient. You can provide short versions to be used in such places, for example, by specifying the following commands:

```
\title[Short title]{Long Informative Title}
\author[Shortened name]{Author's Complete Name}
\date[2015/06/24]{Conference on X at Y, June 24, 2015}
```

The same is possible for the `\institute` and `\subtitle` commands, if you would use these commands in your presentation.

You can provide short names for sections and subsections in exactly the same way, so they would better fit into their field within the frame margin; just use the optional argument in square brackets. The `\part` and `\subsubsection` commands work similarly, they can get a short name in square brackets as well.

Uncovering information piecewise

Showing a complete slide at once may be a bit distracting. People may read ahead instead of listening to you. You can take them by the hand by displaying the content step by step.

The simplest way is to insert a `\pause` command. It can go between anything such as text, graphics, and blocks. It also works between two `\item` commands in a bulleted list; however, consider pausing between whole lists instead of items. Simply use it like we will do in the following line of code:

```
Text\pause more text\pause\includegraphics{filename}
```

Such a frame is then layered; that is, divided into overlays. They are internally numbered. If you would like to show something at a certain overlay, you can tell the `beamer` package when to uncover it:

```
\uncover<3->{Surprise!}
```

This shows your text on slide 3 of the current frame; and it will stay on the following slides in that frame. Omit the dash for restricting it to only slide 3. You can also list multiple slides, such as `<3,5>`, give ranges such as `<3-5>`, and mix the two, as in `<3,5->`.

This syntax works with overlay-specification-aware commands. Among them, there are `\item`, `\includegraphics`, and even `\renewcommand`, so you can use them with an overlay specification such as the following:

```
\includegraphics<3->{filename}
```

Overlays should not be too fancy. A presentation needs a linear structure; complicated overlays may be handy for showing and hiding annotation to an object, while you explain that.

Refer to the `beamer` manual for more information on using overlays. You can open that manual via `texdoc beamer` in Command Prompt, or online at `http://texdoc.net/pkg/beamer`.

Splitting frames in columns

You can arrange text and images in multiple columns on a frame. This is especially handy for images with explanatory text. Let's take a look at a sample:

```
\begin{frame}
  Some text which can use whole frame width
  \begin{columns}[t]
    \begin{column}{0.45\textwidth}
      Sample text in\\
      two lines
    \end{column}
    \begin{column}[T]{0.45\textwidth}
```

```
      \includegraphics[width=3cm]{filename}
    \end{column}
  \end{columns}
\end{frame}
```

We started the multicolumn area using the `column` environment. You can specify alignment options `t`, `b`, or `c` for top, bottom, or centered alignment of the column. Centered is the default. While `t` aligns at the baseline of the first line, which is common in LaTeX, there's a handy additional option, `T`, which aligns at the very top.

Each column is created by the `column` environment. The column width is given as an argument. It understands the same positioning options, so you can override what you set in the surrounding columns environment. We added the `[T]` option here, because an image has its baseline at its bottom, and we want to change it to the very top.

Showing an outline for each section

You can tell the `beamer` package to give an outline at the beginning of each section by specifying the following code:

```
\AtBeginSection{
  \begin{frame}{Outline}
    \tableofcontents[currentsection]
  \end{frame}}
```

You can also use the `\AtBeginSection` command to insert different lines of code. If something should happen in case of a starred `\section*` too, you can insert the corresponding code within an optional argument in square brackets after `\AtBeginSection`.

Removing the navigation symbols

By default, every slide shows small navigation symbols; here, at the bottom of a frame. If you don't need them, you can save that space and reduce the distraction by specifying the following line of code:

```
\setbeamertemplate{navigation symbols}{}
```

Changing the font

The default font set with the `beamer` package is Computer Modern. You can change it to other fonts as discussed in *Chapter 3*, *Adjusting Fonts*.

The default font is sans serif. Even the math formulas are sans serif. With a low projector resolution, or at some distance, this can be more readable than with a serif font.

However, if you would like to change to a serif font, you can load the corresponding font theme in the preamble:

```
\usefonttheme{serif}
```

Another available font theme is `professionalfonts`, which actually doesn't change fonts, but simply uses the set you specify separately. Furthermore, there are `structurebold`, `structureitalicserif`, and `structuresmallcapsserif`, which change the font in the structure, that is, in headlines, footlines, and sidebars, to such a shape combination.

Changing the color

The quickest way to change colors is by loading a theme with a thoughtful selection of colors for the various structure elements. Use a single command as follows:

```
\usecolortheme{dolphin}
```

There are *outer* color themes, providing a color set for headlines, footlines, and sidebars. The author of the `beamer` package gave them sea animal names: `dolphin`, `whale`, and `seahorse`. Then, there are *inner* color themes for elements, such as color blocks, with names of flowers: `lily`, `orchid`, and `rose`. Combine inner and outer color themes as you like.

Finally, there are complete themes covering all structure aspects: `albatross`, `beaver`, `beetle`, `crane`, `dove`, `fly`, `monarca`, `seagull`, `spruce`, and `wolverine`. They are named after flying animals, except `wolverine`, `beaver` and `spruce`, which are external additions.

A lot of names, you may go through them using the `\usecolortheme` command to find the color set you like the most.

Loading a theme from the Internet

With some labor, you can create your very own theme. The extensive `beamer` manual will guide you. However, you may save a lot of time; beamer is very popular among academic users, who already use LaTeX for their papers, so you can find a lot of themes prepared for universities and institutes, but also designed by various beamer users.

You can find an overview of `beamer` themes at `http://latex-beamer.net`.

Explore the gallery there, download the theme you like, add your logo, and tweak it. Instructions for using the themes that are available on the website.

Providing a handout

You can give your audience a printed version of the slides. Just create a version of your document with the `handout` option, so no overlays will be used:

```
\documentclass[handout]{beamer}
```

Slides are commonly small, so it's good to print several slides on a single page:

```
\usepackage{pgfpages}
\pgfpagesuselayout{4 on 1}[a4paper,
  border shrink=5mm,landscape]
```

This prints four slides on an A4 page in landscape. You can get bigger prints, two slides on each page, in portrait mode, by specifying the following line of code:

```
\pgfpagesuselayout{2 on 1}[a4paper,border shrink=5mm]
```

We used the `pgfpages` option, a utility package that comes with the `pgf` package.

See also

For the best quality with included graphics, refer to *Chapter 4*, *Working with Images*. In *Chapter 9*, *Creating Graphics*, you can find recipes for quickly creating diagrams and charts, which are good tools for visualizing data in a presentation.

Designing a curriculum vitae

Tabular layouts are very common today for the **Curriculum Vitae** (**CV**). When applying for a job, inform yourself about the typical requirements of the content of a CV. You can then create a simple document with tables, consistently and clearly readable.

If it needs to be made quickly or if you would like to base it upon a proven modern layout, you can use a template. In this recipe, we will use the `moderncv` class and its template to quickly produce a CV.

Getting ready

If it's not already installed on your computer, download and install `moderncv` from CTAN (`http://ctan.org/tex-archive/macros/latex/contrib/moderncv`).

If your TeX installation provides a package manager, use it for the installation.

There's a directory of examples, containing templates which you can use. Either locate it in the documentation branch of your TeX directory tree, or visit the preceding CTAN link for downloading.

How to do it...

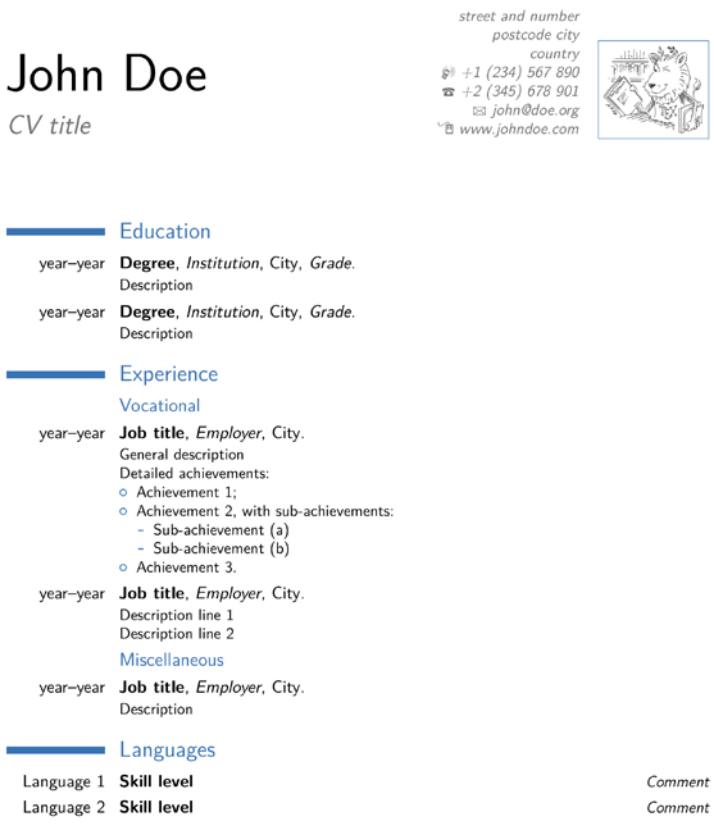
We will start using a sample file provided by the `moderncv` package. Perform these steps:

1. Copy the file `template.tex`, into your document directory and rename it. Choose any name you like. I named it as `MyCV.tex`.
2. Open that document, that is, `MyCV.tex`, and look around to understand the template. Luckily, it is full of comments on how to use it. Compile it to ensure that this document works.
3. Review and edit the class and package options in `MyThesis.tex`.
4. Remove the filler text and type in your own data. At the beginning, your document may look like this:

```
\documentclass[11pt,a4paper,sans]{moderncv}
\moderncvstyle{classic}
\moderncvcolor{blue}
\usepackage[scale=0.75]{geometry}
\name{John}{Doe}
\title{CV title}
\address{street and number}{postcode city}{country}
\phone[mobile]{+1~(234)~567~890}
\phone[fixed]{+2~(345)~678~901}
\email{john@doe.org}
\homepage{www.johndoe.com}
\photo[64pt][0.4pt]{ctanlion.pdf}
\begin{document}
\makecvtitle
\section{Education}
\cventry{year--
year}{Degree}{Institution}{City}{\textit{Grade}}%
  {Description}
\cventry{year--
year}{Degree}{Institution}{City}{\textit{Grade}}%
  {Description}
\section{Experience}
\subsection{Vocational}
\cventry{year--year}{Job title}{Employer}{City}{}%
  {General description\newline{}Detailed achievements:%
\begin{itemize}%
  \item Achievement 1;
```

```
    \item Achievement 2, with sub-achievements:
      \begin{itemize}%
      \item Sub-achievement (a)
      \item Sub-achievement (b)
    \end{itemize}
  \item Achievement 3.
  \end{itemize}}
  \cventry{year--year}{Job title}{Employer}{City}{}
    {Description line 1\newline{}Description line 2}
  \subsection{Miscellaneous}
  \cventry{year--year}{Job
  title}{Employer}{City}{}{Description}
  \section{Languages}
  \cvitemwithcomment{Language 1}{Skill level}{Comment}
  \cvitemwithcomment{Language 2}{Skill level}{Comment}
  \end{document}
```

5. Compile and take a look at the following result:



How it works...

We loaded the `moderncv` package. We used 11 pt as the base font size; 10 pt and 12 pt are also supported. We selected A4 paper; other paper size options are `a5paper`, `letterpaper`, `legalpaper`, and `executivepaper`. You can also add the `landscape` option. We chose a sans serif font, which is fine for such lists; alternatively, you could write Roman alphabets for a serif font.

We selected the `classic` style. Other available styles are `casual`, `oldstyle`, and `banking`.

Our color style is `blue`. Other color options are `orange`, `green`, `red`, `purple`, `gray`, and `black`.

We loaded the `geometry` package with a scaling factor for reducing the margins.

Using macros such as the `\name` and `\address`, we entered our personal data.

The `\photo` command includes our photo; the size options are the height to which it is scaled and the thickness of the frame around the photo. In this recipe, we used the CTAN lion drawn by Duane Bibby for the photo.

The document body is divided into sections and subsections, just with a special design.

Then, the `\cventry` command makes a typical resume entry for job or education. Use it as follows:

```
\cventry[spacing]{years}{job title}
  {employer}{localization}{detail}{job description}
```

Otherwise, you can use the following:

```
\cventry[spacing]{years}{degree}
  {institution}{localization}{grade}{comment}
```

You can leave the last four arguments empty if you don't need them.

A simpler line is created using `\cvitem`, as follows:

```
\cvitem[optional spacing length{header}{text}
```

The `\cvitemwithcomment` command works in a similar way, just with another argument that is printed at the right.

If you are looking for deeper information beyond this quick start guide, some more commands and options are explained in the very well-documented `template.tex` file, as well as in the class file `moderncv.cls` itself.

See also

The template file contains a letter template that you can use for an application. Another approach is explained in the next recipe.

Writing a letter

Letters have a specific structure. Commonly, they have an addressee field at a fixed position, which should be visible in the envelope window. It also should show a back address, of yourself as the sender. An opening text and a closing phrase are usual elements, and you may add fold marks and enclosures.

How to do it...

We will use a KOMA-Script class, which has been specifically designed for letters, named `scrlttr2`. Take a look at the following steps:

1. Use the `scrlttr2` class, activate the address field and fold marks using the options as follows, and align the sender's address to the right:

```
\documentclass[addrfield=true,foldmarks=true,
  fromalign=right]{scrlttr2}
```

2. Provide your name and your address using the `\setkomavar` command:

```
\setkomavar{fromname}{Thomas Smith}
\setkomavar{fromaddress}{123 Blvd \\ City, CC 12345}
```

3. Write a date, either `\today` for today, or any date as text:

```
\date{\today}
```

4. Begin the document:

```
\begin{document}
```

5. Open a `letter` environment, with the recipient's address as an argument:

```
\begin{letter}{Agency \\ 5th Avenue \\
               Capital City, CC 12345}
```

6. Start with an opening, and let your letter text follow:

```
\opening{Dear Sir or Madam,}
the actual content of the letter follows.
```

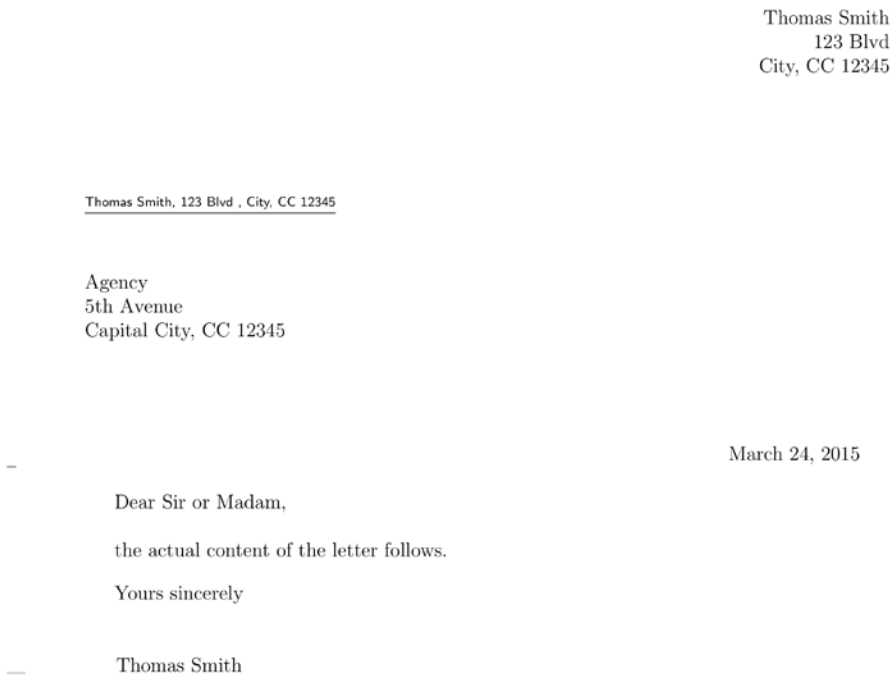
7. End with closing words:

```
\closing{Yours sincerely}
```

8. End the `letter` environment and the document:

```
\end{letter}
\end{document}
```

9. Compile the document. Here is the upper part of the output:



That was pretty easy! You got a fully fledged formal letter with addressing information, an envelope window support, date of writing, phrases, signature, and even fold marks.

Now you can enter the real addresses and the actual letter text.

How it works...

When loading the letter class `scrlttr2`, we activated the address field, switched on fold marks, and set the options for aligning the sender's address to the right.

The `scrlttr2` class is quite different from other classes, so it has a special interface. Using the `\setkomavar` command, we set the content of class variables, similar to the `\renewcommand` command. Here, we set the name and address. The KOMA-Script manual explains all the available variables. As mentioned in the recipe *Writing a book*, you can open it by typing the `texdoc scrguien` command in Command Prompt, or online at `http://texdoc.net/pkg/scrguien`.

We used a `letter` environment for the actual content, including the opening and closing phrases. The address is a mandatory argument for that environment. You can have several `letter` environments in a single document.

There's more...

For improving input and hyphenation, and for changing the font, take a look at the first recipe in *Chapter 2*, *Tuning the Text*.

Let's take a look at some letter specific options.

Separating paragraphs

Instead of indenting the beginnings of paragraphs, you can indicate a paragraph break with an empty line instead. For this, simply add the following option to the comma-separated list of class options at the beginning:

```
parskip=full
```

Use the `parskip=half` option for less space between paragraphs.

Changing the signature

If you would like to use a signature that is different from your specified name for the address, you can modify the corresponding variable content in the preamble:

```
\setkomavar{signature}{Thomas}
```

It would be indented. You can get it left-aligned by specifying the following code:

```
\renewcommand{\raggedsignature}{\raggedright}
```

The preceding code also belongs to the preamble.

Adding enclosures

If you wish to add enclosures to your letter, it's a common practice to mention them. You can do this by inserting an `\encl` command right before the `\end{letter}` command:

```
\encl{Curriculum vitae, certificates}
```

You can change the default `encl:` if you like by modifying the corresponding variable before calling the `\encl` option:

```
\setkomavar*{enclseparator}{Attached}
```

We used the starred version, `\setkomavar*`, which modifies the description of a variable instead of its content, which actually is `:`, that is, a colon followed by a space.

Producing a leaflet

Flyers are a common way to promote an event or to inform about a product. In particular, a folded leaflet is very handy as a giveaway and to carry around, so let's see how to produce one.

How to do it...

The intended layout is very different compared to the already shown document types. Fortunately, there's a document class for it with the name `leaflet`, which we will use now. Let's start filling it with some content. Let's take a look at the following steps:

1. Start with the `leaflet` document class. Choose a base font size of 10 pt, and set the option `notumble`, which keeps the back side printed in the same direction:
   ```
   \documentclass[10pt,notumble]{leaflet}
   ```
2. Use an extended font encoding:
   ```
   \usepackage[T1]{fontenc}
   ```
3. If you want a nondefault font, load it, this time we chose the **Linux Libertine** font:
   ```
   \usepackage{libertine}
   ```
4. Switch to sans serif as the default font family:
   ```
   \renewcommand{\familydefault}{\sfdefault}
   ```
5. For better text justification, load the `microtype` package:
   ```
   \usepackage{microtype}
   ```
6. Load the `graphicx` package to include a picture:
   ```
   \usepackage{graphicx}
   ```
7. Switch off the page numbering:
   ```
   \pagenumbering{gobble}
   ```
8. Begin the document:
   ```
   \begin{document}
   ```
9. Set the title, author's name, and date. Then, print the title:
   ```
   \title{\textbf{\TeX\ Live Install Party}}
   \author{\Large\textbf{Your \TeX\ team}}
   \date{\textbf{August 11, City Hall Cellar}}
   \maketitle
   ```

10. Include a centered image. Here, we chose the CTAN lion:

```
\begin{center}
  \includegraphics[width=\linewidth]{ctanlion.pdf}
\end{center}
```

11. Add some text:

```
We'd like to welcome you to our famous yearly \TeX\ install
party! Bring your laptop and have free cold soft drinks
while we assist you in installing the latest \TeX\ version
on your computer.
We will provide
```

12. A bulleted list can be a good idea for a catchy text. Use an `itemize` environment for it, each list item starting with `\item`:

```
\begin{itemize}
  \item a fast internet connection fow downloading,
  \item media such as DVDs and USB sticks with
    the latest \TeX\,
  \item \TeX\ books for bying with a discount,
  \item chat with \TeX\ experts.
\end{itemize}
```

13. End the page. Fill in more text on the next page of the leaflet, which will be printed on the back side of the paper, next to each other:

```
\clearpage
Fill in text for page 2 (on the back side)
\clearpage
Fill in text for page 3 (on the back side)
\clearpage
Fill in text for page 4 (on the back side)
\clearpage
```

14. Now, pages 5 and 6 of the leaflet come to the front side of the paper. Use the `\section` command to insert a heading:

```
\section{Schedule}
```

15. You can add a timetable using a `tabular` environment. Using the `@{}` option, we suppress spacing at the left and right:

```
\begin{tabular}{@{}rl@{}}
  6 pm    & Welcome \\
```

```
  7:30 pm & Live install presentation \\
  8 pm    & Book authors available for talks and signing \\
  9:30 pm & Bar closing
\end{tabular}
```

16. End the paragraph with an empty line, continue writing the text, and finally end the document:

```
From 6pm to 10pm: install support and free \TeX\ copies on DVD on
our welcome desk.
\section{Accomodation}
Hotel, Meals, Travel information here
\section{Sponsors}
Information about our local \TeX\ user group
and Open Source projects sponsor
\clearpage
\section{Contact}
Names, Phone numbers, email addresses
\end{document}
```

17. Compile and take a look at the first page:

Schedule

6 pm Welcome
7:30 pm Live install presentation
8 pm Book authors available for talks and signing
9:30 pm Bar closing

From 6pm to 10pm: install support and free TEX copies on DVD on our welcome desk.

Accomodation

Hotel, Meals, Travel information here

Sponsors

Information about our local TEX user group and Open Source projects sponsor

Contact

Names, Phone numbers, email addresses

TEX Live Install Party

Your TEX team

August 11, City Hall Cellar



We'd like to welcome you to our famous yearly TEX install party! Bring your laptop and have free cold soft drinks while we assist you in installing the latest TEX version on your computer.

We will provide

- a fast internet connection fow downloading,
- media such as DVDs and USB sticks with the latest TEX
- TEX books for bying with a discount,
- chat with TEX experts.

The back side still contains just some dummy text, which helps to identify the position where the text finally lands on the page.

How it works...

In the first line, we loaded the `leaflet` package with a font size of 10 pt. The `notumble` option suppresses the default behavior, that is, printing the back side upside down.

The next three lines contain our font settings. We used the Linux Libertine font and specified the `T1` font encoding. You can read more about encodings in *Chapter 2*, *Tuning the Text*, specifically in the *Improving justification and hyphenation* recipe. Furthermore, we set the default font family to be sans serif. I prefer the clean look of sans serif on a flyer or a leaflet, which usually contains little text.

The remaining part of the preamble is as follows:

- We loaded the `microtype` package, which improves the justification capabilities with tiny font adjustments. This is especially useful in a situation with narrow columns, such as in this case.
- We loaded the `graphicx` package, so we are able to include images such as a logo or a geographic map.
- We hid the page numbers. The `\gobble` command is a TeX command that removes the following command or control sequence, so the page number will simply be absorbed.

Our document body shows usual sectioning commands and text. You can see that we added an explicit space after the TeX logo by inserting a backslash and a following space. That's because the space after a macro, such as the `\TeX` command, just indicates the end of the macro. It doesn't produce a space in print because punctuation may follow the macro.

To have an image in our template, we used the CTAN lion drawn by Duane Bibby; simply replace it with your own image, for example, a geographic map or logo.

The remaining text is straightforward and shows some useful layout details, as follows:

- Centering using the `center` environment:

```
\begin{center}
...
\end{center}
```

- Arranging points in a bulleted list using an `itemize` environment
- Setting up a `tabular` environment for text, which should be aligned in columns

In the `\begin{tabular}{@{}rl@{}}` line, the characters `rl` stand for two columns, where the first one is right aligned and the second one is left aligned. The expression `@{code}` inserts a piece of code instead of the space before or after the column, so `@{}` replaces it with nothing, that means, removes it. We got two columns without additional whitespaces on the left or right, saving our previous line space.

There's more...

The `leaflet` class provides some options and commands for customization.

Fold marks and cut lines

By default, a small folding mark is printed on the back side. If you would like to omit it, add the `nofoldmark` option when loading the class:

```
\documentclass[10pt,notumble,nofoldmark]{leaflet}
```

You can draw a vertically dotted line with a scissors symbol using the `\CutLine` command in the preamble with a page number as an argument. The line will go between this one and the preceding page, which is as follows:

```
\CutLine{3}
```

This will print a dotted line with two scissors symbols on the back side, between pages 2 and 3, where a folding mark will be placed by default. The starred command version `\CutLine*` will not print the scissors.

Adjusting the margins

Similar to standard classes, you can use page headers and footers. There are none by default here. Standard commands such as `\setlength{\headheight}{...}` and `\pagestyle` can be used. The `leaflet` provides an additional command to declare the margins:

```
\setmargins{top}{bottom}{left}{right}
```

Adding a background image

You can add an image to the background of a certain page:

```
\AddToBackground{pagenumber}{\includegraphics{filename}}
```

Use the starred version `\AddToBackground*` to allow it to be printed in the background of the combined page.

Instead of the `\includegraphics` command, you can use other positioning commands, including our drawing code. Here, the *Absolute positioning of text* recipe in *Chapter 2*, *Tuning the Text*, will be useful.

Changing the sectioning font

The font size of section headers is already a bit smaller than that of standard classes. If you would like to change the size, shape, or color of the headings, you can redefine the `\sectfont` macro. For example, if we enable using color with the `\usepackage{xcolor}` command, we can write the following line of code:

```
\renewcommand{\sectfont}{\large\sffamily\bfseries\color{blue}}
```

This will give a large sans serif font in bold and with blue color.

For further information regarding fonts, refer to *Chapter 3*, *Adjusting Fonts*.

Creating a large poster

We have seen informational or scientific posters at conferences or on the walls of universities or institutes. They mostly have certain characteristics in common:

- They have a large size, such as A2, A1, or even A0
- People may look at them from far away, but also from a very short distance

In consequence, we get some requirements for typesetting:

- The page layout dimensions should work with such a big size.
- We need a wide range of font sizes. We should be able to read while standing close, but we also need large, catchy headings.
- The poster should be partitioned into digestible blocks. Specifically, each block should not exceed the usual line width we know from body texts. Also, excessively wide lines will make it hard to focus and to skip back to the start of the next line. So, the lines in blocks should not be much wider than about 40 or 50 characters long.
- Blocks should have distinct headings.
- Graphical elements such as colors and lines can be used to divide the poster contents in parts.
- Images should be vector graphics or should have a high resolution.

In this recipe, we will create a poster of A0 size in landscape orientation. It will show some blocks containing the dummy text as a placeholder, math, and images. As sample images, we will take a flowchart from *Chapter 9*, *Creating Graphics*, and a plot from *Chapter 10*, *Advanced Mathematics*. There, you can find the source code. You can later replace the dummy text and other parts with your own content.

How to do it...

We will use the `tikzposter` class. The document is structured in columns and blocks. Follow these steps:

1. Begin with the `document` class. A0 is the default paper size. We state landscape orientation as the option:

   ```
   \documentclass[landscape]{tikzposter}
   ```

2. Choose a theme, which provides a set of colors and decorations. We choose the blue `Wave` theme:

   ```
   \usetheme{Wave}
   ```

3. Load the `lipsum` package to generate the dummy text:

   ```
   \usepackage{lipsum}
   ```

4. For dividing wider blocks in text columns, we load the `multicol` package. On account of the large paper, we set the column separation and the separation line width to high values:

   ```
   \usepackage{multicol}
   \setlength{\columnsep}{4cm}
   \setlength{\columnseprule}{1mm}
   ```

5. Define macros for repeated tasks, if desired. In our case, it will be a macro for including an image with an optional caption:

   ```
   \newcommand*{\image}[2][]{%
     \begin{tikzfigure}[#1]
       \includegraphics[width=\linewidth]{#2}
     \end{tikzfigure}}
   ```

6. Start the document:

   ```
   \begin{document}
   ```

7. Declare the author and title, and print it out:

   ```
   \title{\LaTeX\ in Use}
   \author{John Doe}
   \maketitle
   ```

8. Begin a set of columns:

   ```
   \begin{columns}
   ```

9. Start a column with a width of 65 percent of the available text width:

```
\column{.65}
```

10. Define a block with the title `Workflow` in the first argument; the second argument containing dummy text and an image:

```
\block{Workflow}{
  \lipsum[1]
  \image[\LaTeX\ workflow]{flowchart}
}
```

11. Still in the first column, start a set of subcolumns:

```
\begin{subcolumns}
```

12. The first subcolumn will take half of the available width, in this case, the width of the left column:

```
\subcolumn{.5}
```

13. Create a block with a bulleted list and a mathematical equation to get a feeling of how it will look on a poster. We will also use a colored box and an inner block with a title for the equation:

```
\block{Mathematics}{
  Take a coffee, then:
  \bigskip
  \coloredbox{\begin{itemize}
    \item State
    \item Proof
    \item Write in \LaTeX
  \end{itemize}}
  \bigskip
  \innerblock{Integral approximation}{
    \[
      \int_a^b f(x) dx \approx (b-a)
        \sum_{i=0}^n w_i f(x_i)
    \]
  }
}
```

14. Add a note, which will have a callout shape, pointing to the formula:

```
\note[targetoffsetx = 4.5cm, targetoffsety = -5cm,
  angle = -30, connection]{Weight function}
```

15. Make another subcolumn, taking the other half of the available width. Insert a block filled with text and end the `subcolumns` environment:

```
    \subcolumn{.5}
    \block{Text}{\lipsum[1]}
  \end{subcolumns}
```

16. Now that we are back to our main `column` environment, make another column, print a block with an image and some text, and then end the `columns` environment:

```
  \column{.35}
  \block{Plotting functions}{
    \image{plot}
    \lipsum[4]
  }
\end{columns}
```

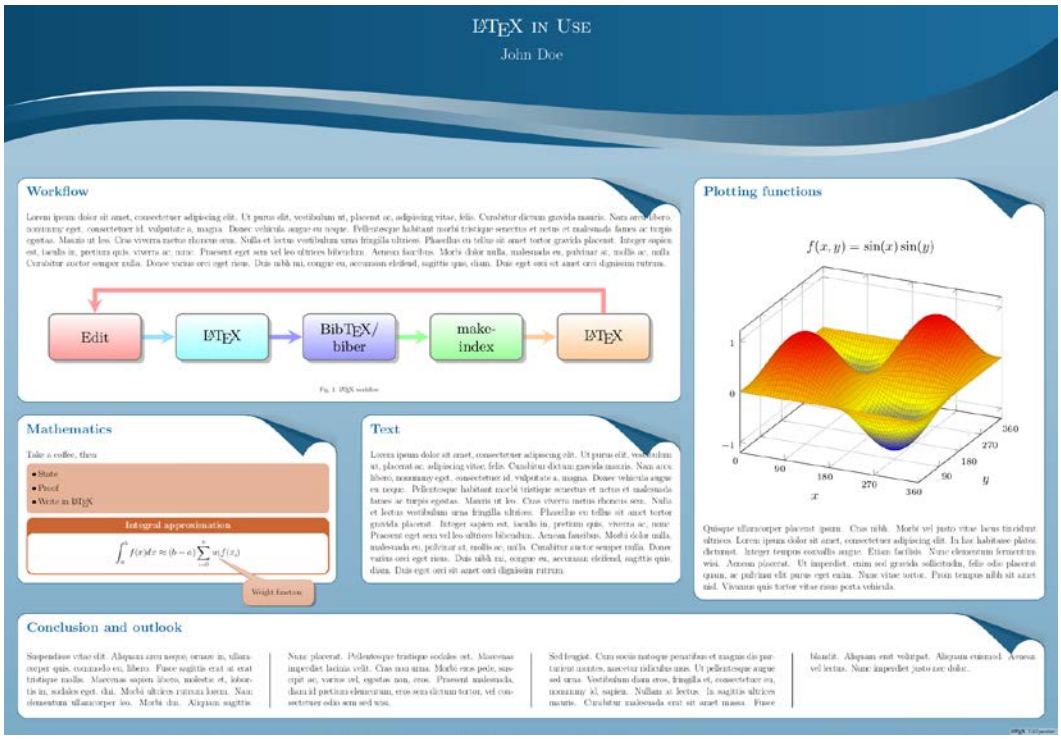
17. As we ended the columns, a block will use the whole available width. To keep the text readable, we will now use the `multicol` package. We divide the text itself into columns using a `multicolumn` environment with four columns:

```
\block{Conclusion and outlook}{
  \begin{multicols}{4}
    \lipsum[10-11]
  \end{multicols}
}
```

18. End the document:

```
\end{document}
```

19. Compile the document, and take a look:



How it works...

The `tikzposter` package supports large paper sizes, large fonts, and it takes care of block heights and spacing between columns. We, as users, just decided the relative column width.

Several class options are provided. You can add them to the `\documentclass` command like we did with the preceding `landscape` option. Let's take a look at the following:

- The paper size can be chosen with the `a0paper`, `a1paper`, or `a2paper` options. The `a0paper` option is the default.
- The available font sizes are `12pt`, `14pt`, `17pt`, `20pt`, and `25pt`. The last one is the default.
- You can select the orientation by using the `landscape` or `portrait` command. The `portrait` option is the default.
- The standard option `fleqn` for flush left equations is supported.
- The standard option `leqno` for numbering the equation at the left-hand side is also supported.

You can adjust several lengths using different options. Give them as class options in the `key=value` form with a measurement unit such as mm or cm:

- `margin`: This is the distance between the edge of the poster area and the edge of the paper
- `innermargin`: This is the distance from the outermost edge of the blocks to the edge of the poster
- `colspace`: This is the horizontal distance between consecutive columns
- `subcolspace`: This is the horizontal distance between consecutive columns in a subcolumn environment
- `blockverticalspace`: This is the distance between the bottom of a block and the top of the next block

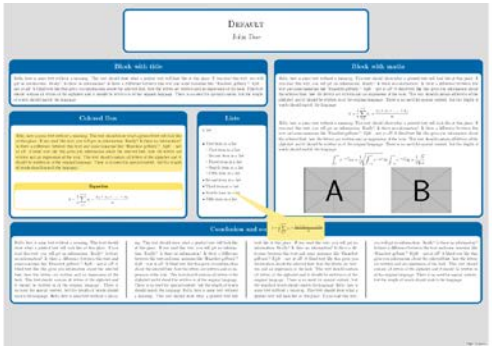
A sample call using the defaults will be as follows:

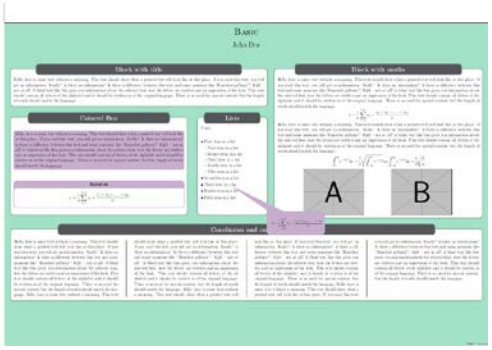
```
\documentclass[a0paper, portrait, 25pt, margin=0mm,
  innermargin=15mm, colspace=15mm, subcolspace=8mm,
  blockverticalspace=15mm]{tikzposter}
```

The `tikzposter` package makes use of the very capable graphics language **TikZ**. We will see more of TikZ in *Chapter 9*, *Creating Graphics*. For now, the main benefit is that `tikzposter` provides a lot of predefined styles and color schemes.

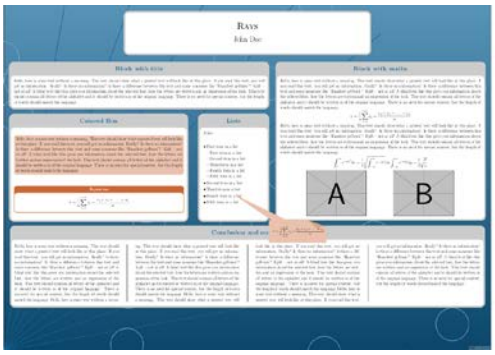
You can use a main layout style by using the `\usetheme{name}` command. When this book came out, there were nine themes available:

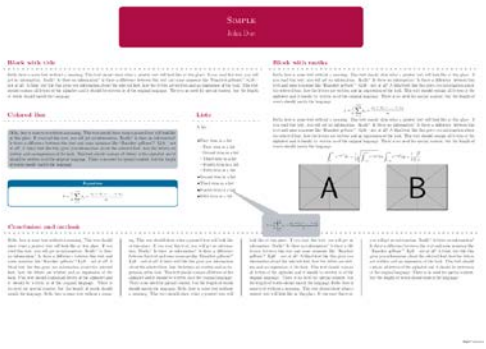
- `Wave`: This can be seen in our preceding recipe
- `Default` (left) and `Basic` (right): This is shown in the following image:



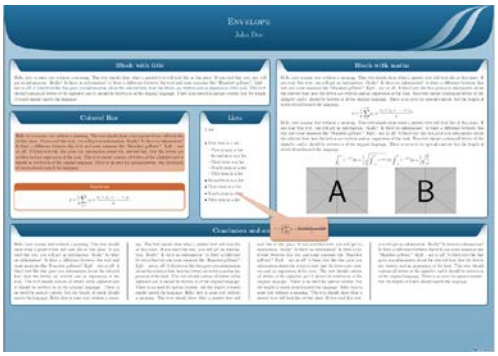


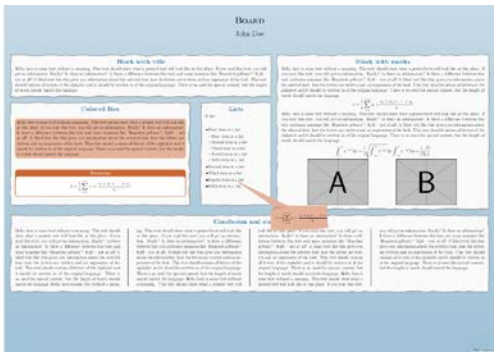
- `Rays` (left) and `Simple` (right), as shown in the following image:



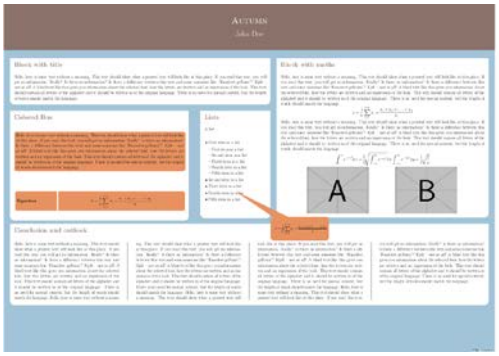


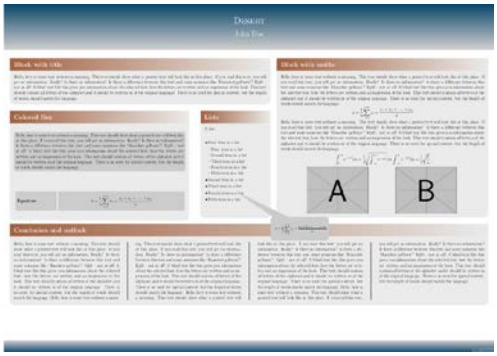
- `Envelope` (left) and `Board` (right), as shown in the following image:





- `Autumn` (left) and `Desert` (right), as shown in the following image:





Furthermore, there are predefined styles for color, title, background, notes, blocks, and inner blocks, which can be chosen and composed. There's support for creating further styles.

The commands, which you have seen previously, can be used without support options straight away. However, they can be customized using several options.

The full reference is available by typing the `texdoc tikzposter` command in Command Prompt, and online at `http://texdoc.net/pkg/tikzposter`. You can find a style guide, a feature guide, and much more at `https://bitbucket.org/surmann/tikzposter/downloads`.

There's more...

One of the first poster classes is `a0poster`. It actually supports the paper sizes A0, A1, A2, and A3. It provides font sizes from 12 pt up to 107 pt. Math formulas are printed in a suitable size. There's no specific support for graphics, color, or text placement. For this, you would need additional packages, such as TikZ.

In the recipe *Creating a presentation*, you saw the `beamer` package as a presentation class. The `beamerposter` package can be used together with it to produce presentations in poster size. It combines the `beamer` package with the `a0poster` code. So, you can produce large posters with a wide range of font sizes together with the `beamer` package's color and graphics capabilities, such as the `beamer` boxes with titles.

As mentioned previously, you can use the `texdoc` command or the Internet site `http://texdoc.net` to access the documentation of the aforementioned classes and packages.

Another solution is provided by the `baposter` template. It provides blocks with headings and positioning support. Furthermore, it offers a set of predefined styles. Its download and documentation are available at `http://www.brian-amberg.de/uni/poster/`.

2
Tuning the Text

This chapter contains the following recipes:

- Inputting accented characters
- Improving justification and hyphenation
- Converting numbers to words
- Putting text into a colorful box
- Visualizing the layout
- Visualizing boxes of letters and symbols
- Typesetting in a grid
- Absolute positioning of text
- Starting a paragraph with an oversized letter
- Fitting text to a shape
- Creating a pull quote

Introduction

The previous chapter presented recipes for creating whole documents. You may use them as starting points. Now, we will focus on customizing text details. The upcoming sections broadly deal with the following:

- Manipulating text
- Positioning and arranging text
- Shaping paragraphs

We are not yet dealing with fonts. This topic deserves a whole chapter, and will be covered in *Chapter 3*, *Adjusting Fonts*.

We will start with some very useful basics, cover some helpful things, and end the chapter with recipes that show off what LaTeX can do beyond rectangular paragraphs.

Inputting accented characters

By default, LaTeX works with simple ASCII characters. For accented characters, such as in the German word "schön", you need to type `sch\"on` into your editor. The `babel` command with the `ngerman` option simplifies the syntax to `sch"on`. But there is an easier way.

How to do it...

We will activate extended input character support as follows:

1. Check your editor's configuration and find out its input encoding setting. `utf8` (UTF-8 means Unicode) is standard for Linux, Mac OS X, and some Windows editors. However, some Windows editors still work with `latin1` or `cp1250/cp1252`, while some older Macs use `applemac`.
2. Load the `inputenc` package with the corresponding option, like so:

   ```
   \usepackage[utf8]{inputenc}
   ```

3. Now you can directly type characters such as *a*, *ü*, *ö*, *é*, and *è* in to your document.



If you need help with editor settings, you can post a question in the forum at `http://latex-community.org/forum/`. This web site provides support forums for various LaTeX editors.

How it works...

The `inputenc` package enables direct entering of accented characters and symbols from code tables other than ASCII, and of Unicode. Modern editors and operating systems support UTF-8, so `utf8` should be preferably set in the editor.



There's an `utf8x` encoding, but beware: this is not an extension, but a different approach, useful for some Asian languages.

Be careful: the `inputenc` encoding option, the editor setting, and the actual file encoding of the operating system all need to match.

There's more...

If you would like to change the existing encoding, such as for files produced on a different system, you could open such a file in your editor with the actual encoding and save it with the new encoding. You may also use a conversion program such as one of these:

- `recode`: See `http://latex-community.org/tools/recode`
- `iconv`: See `http://latex-community.org/tools/iconv`

There's also a clever way of automatically selecting the needed encoding by using the `selinput` package as follows:

1. Load the `selinput` package:
   ```
   \usepackage{selinput}
   ```
2. Specify some meanings of characters via this command; for example, as follows:
   ```
   \SelectInputMappings{
     adieresis={ä},
     eacute={é},
     germandbls={ß}}
   ```
3. In your document text, you can now type *ä*, *é*, or *ß*, and they will be printed correctly.

Here, we provide samples of required characters with their meanings, and the `selinput` package determines the matching encoding. A few chosen glyphs may already suffice. If you would like to specify more, you can find their names in the `selinput` manual. It can be opened by typing the `texdoc selinput` command in Command Prompt, or accessed online at `http://texdoc.net/pkg/selinput`.

In addition to not needing to deal with encodings, you can also easily change the editor or operating system. You can exchange your files with friends who may use a different encoding without changing any setting.

Improving justification and hyphenation

Sometimes, you may get warnings like overfull `\hbox`, or you may notice words hanging in the margin. This is a sign that LaTeX has had serious problems with justification. Now, we will take a look at how to improve that.

How to do it...

You can start with any document. We will optimize it with settings in the preamble. If you don't have one at hand, you can take one from the code package for this book, specifically for the first chapter, or download one at `http://latex-cookbook.net`. Let's proceed:

1. Load the package `babel` with your document languages as options. Use the preferred language as the last option:

   ```
   \usepackage[ngerman,english]{babel}
   ```

2. If you would like to use handy shortcuts of the `ngerman` package with English too, add the following lines of code:

   ```
   \useshorthands{"}
   \addto\extrasenglish{\languageshorthands{ngerman}}
   ```

3. Load the `fontenc` package with the `T1` option set:

   ```
   \usepackage[T1]{fontenc}
   ```

4. Load the `microtype` package for improved micro-typography:

   ```
   \usepackage{microtype}
   ```

How it works...

With the `babel` package, LaTeX uses the hyphenation patterns for the chosen language. Even for the English language it could be useful. As you have seen, the `babel` package can additionally load the handy `ngerman` shortcuts for hyphenation commands. You can find such shorthand commands in the `babel` manual. You can open it by typing the `texdoc babel` command in Command Prompt, or you can find it online at `http://texdoc.net/pkg/babel`.

The modern font encoding T1 improves the situation further. When TeX and LaTeX were introduced, fonts did not contain glyphs for accented characters. They were printed as two characters, one being the actual accent. That's bad for copying and pasting from the final PDF output. It also interfered with hyphenation.

The default encoding is called **OT1**, and covers 128 glyphs, which means that it encodes 128 characters. T1 provides 256 glyphs, so many accented characters are included. That's why T1 is recommended for Western European languages. There are further encodings for other languages, such as for Cyrillic and for Asian languages. If you would like to work with them, take a look at the `fontenc` manual by typing the `texdoc fontenc` command in Command Prompt, or you can read it online at `http://texdoc.net/pkg/fontenc`.

Finally, we let the `microtype` package apply micro-typographic extensions. For example, the font size, even of single letters, might be scaled a little bit in favor of better full justification. You hardly can notice this with the naked eye, but you can see the effect of less hyphenation and better overall greyness of the text. Therefore, there should be smaller white gaps between words.

Furthermore, the `microtype` package subtly adjusts punctuation at the margin for better optical alignment, instead of just mechanical justification.

Converting numbers to words

Numbers are sometimes written in words rather than using numerals. LaTeX is capable of automatically converting numbers to words. This is especially useful for values which originate from LaTeX counters, such as page or section numbers.

How to do it...

Load the `fmtcount` package and use its commands for conversion:

1. Start with any document class, such as `article`:

   ```
   \documentclass{article}
   ```

2. Load the `fmtcount` package:

   ```
   \usepackage{fmtcount}
   ```

3. Begin the document:

   ```
   \begin{document}
   ```

4. Write some text. Use the following instructions:

   - Whenever you'd like to convert a number to a word, use the command `\numberstringnum`.
   - To print a counter value as a word, use `\numberstring`.
   - For a similar purpose, but in ordinal form, use `\ordinalstringnum` or `\ordinalstring`. Let's take a look at the following commands:

   ```
   This document should have \numberstringnum{32}
   pages, now we are on page \numberstring{page}
   in the \ordinalstring{section} section.
   ```

5. End the document:

   ```
   \end{document}
   ```

6. Compile the document. All numbers and counters are displayed as text, so you will get the following output:

   ```
   This document should have thirty-two pages. Now we are on page one
   in the first section.
   ```

How it works...

The `fmtcount` package provides a set of commands that print out numbers as words:

- `\ordinalstring{counter}`: This prints the value of a counter as an ordinal in text
- `\numberstring{counter}`: This prints the counter's value as text
- `\ordinalstringnum{number}` and `\numberstringnum{number}`: These two commands do the same job, but based on an actual number instead of a counter
- `\Ordinalstring{counter}`, `\Numberstring{counter}`, `\Ordinalstringnum{number}`, and `\Numberstringnum{number}`: These are the capitalized versions; they print the initial letter in uppercase

There's more...

The `fmtcount` package has even more to offer, such as multilingual capabilities, gender support, and support for new enumeration styles. We shall take a look at these features now.

Multilingual support

The `fmtcount` package supports several languages, such as English, Spanish, Portuguese, French, German, and Italian. It tries to detect the language option that has already been passed to the `babel` or `polyglossia` packages. You can explicitly load required definitions by typing the `\FCloadlang{language}` command in the preamble, with a language name understood by the `babel` package.

Gender

All the preceding commands support an optional gender argument at the end, taking one of the following values: `m` (masculine), `f` (feminine), or `n` (neuter). It is used such as in the `\numberstring{section}[f]` command. `Masculine` is the default.

Enumerated lists

The `moreenum` package provides new enumeration styles based on the `fmtcount` package. Here's an example:

```
\documentclass{article}
\usepackage{moreenum}
\begin{document}
\begin{enumerate}[label=\Nthwords*]
  \item live
  \item long
  \item prosper
```

```
\end{enumerate}
\end{document}
```

The items will be numbered *first*, *second*, and *third*, together with a dot. A label command `\Nwords*` would print `One`, `Two`, `Three`, with a dot. Also, here, there are lowercase versions, starting with a small letter *n*.

Besides the `fmtcount` command, this builds on the `enumitem` package, and so provides a similar `key=value` interface.

Putting text into a colorful box

You often see important content put into a colored box, especially on posters and slides, although it's used in other documents, too. In this recipe, we will put a little text and also whole paragraphs into a colored box. They can also have titles.

How to do it...

We will use the `tcolorbox` package. It is based on `pgf`, so you need to have that package installed as well.

We will create a box with the defaults, a titled box with split content, and boxes placed inline that fit the width of the content. Proceed as follows:

1. Create a small document based on any document class. The `article` package is a simple choice. Load the `blindtext` package to generate dummy text. This time, we will use the `pangram` option to create short pangrams as dummy text. The `blindtext` package requires the `babel` package, so we load it before. We also set English as the language. Furthermore, load the `tcolorbox` package. Our base document looks like this:

```
\documentclass{article}
\usepackage[english]{babel}
\usepackage[pangram]{blindtext}
\usepackage{tcolorbox}
\begin{document}
\end{document}
```

2. Create a simple box by inserting this code right after the `\begin{document}` command:

```
\begin{tcolorbox}
  \blindtext
\end{tcolorbox}
```

This gives you this box with text:

> The quick brown fox jumps over the lazy dog. Jackdaws love my big Sphinx of Quartz. Pack my box with five dozen liquor jugs. The five boxing wizards jump quickly. Sympathizing would fix Quaker objectives.

3. Change it to, or add, the following code snippet:

```
\begin{tcolorbox}[title=\textbf{Examples},
  colback=blue!5!white,colframe=blue!75!white]
  The text below consists of pangrams.
  \tcblower
  \blindtext[3]
\end{tcolorbox}
```

The new box has a title and is divided into two parts, as shown here:

Examples

The text below consists of pangrams.

The quick brown fox jumps over the lazy dog. Jackdaws love my big Sphinx of Quartz. Pack my box with five dozen liquor jugs.

4. Now try this setting and box command in the same document:

```
\tcbset{colframe=green!50!black,colback=white,
  colupper=green!30!black,fonttitle=\bfseries,
  center title, nobeforeafter, tcbox raise base}
  Normal text \tcbox{Boxed text}
```

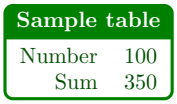
It results in this:

Normal text Boxed text

5. Then, try this variation:

```
\tcbox[left=0pt,right=0pt,top=0.5ex,bottom=0pt,boxsep=0pt,
  toptitle=0.5ex,bottomtitle=0.5ex,title=Sample table]{
  \begin{tabular}[t]{rl}
    Number & 100 \\
    Sum    & 350
  \end{tabular}}
```

You can see that we can create boxes with titles, even with nontrivial content, such as tabular material:



How it works...

We loaded the `blindtext` package with the `pangram` option, which gives us short sample text snippets. We used them as dummy box content.

We loaded the `tcolorbox` package. The LaTeX environment with the same name then generates boxes for us.

The first box was done without any option with default settings. In the next box, we used the `key=value` interface to specify a title and the colors for the frame and the background. The `\tcblower` command split the box for us with a dashed line.

Then, we used the `\tcbset` command to define options which are valid for all following boxes. It's a good idea to use it in the preamble when all boxes would be similar. We set the color, applied a bold typeface, and centered the title. The `nobeforeafter` command implies no additional space before and after the box. The `raise base` command raises the whole box so that the baseline of the content matches the outer baseline. You will notice this when you look at our last boxes.

The last boxes were created using the `\tcbbox` command. It understands most options of the `tcolorbox` package; however, there are two differences:

- `\tcbbox`: It doesn't provide a lower part, unlike the `tcolorbox` package with the `\tblower` command
- `\tcbbox`: Using this command cannot be broken across pages, in contrast to the `tcolorbox` package

The `\tcbbox` command is a good choice for boxes within text. As in our example, it works fine around tabular code. It also works with images.

There's more...

The `tcolorbox` package is extremely capable and provides a lot of options and styles. The latter are also called **skins**. Refer to the package manual to explore further details. The wealth of its features is described over more than 300 pages. You can open the manual by typing the `texdoc tcolorbox` command into the command line or by visiting `http://texdoc.net/pkg/tcolorbox`.

The `mdframed` package offers a similar approach. Based on the classic `framed` package, it works with modern graphics packages such as **TikZ** and **PSTricks** to create colored boxes which can be broken across pages.

Visualizing the layout

At the time of designing a document, one may like to have an exact view of the dimensions and position of the text body, the header and footer, and the space for the margin notes. LaTeX can print help lines for you so you can examine the layout and use it to position extra elements.

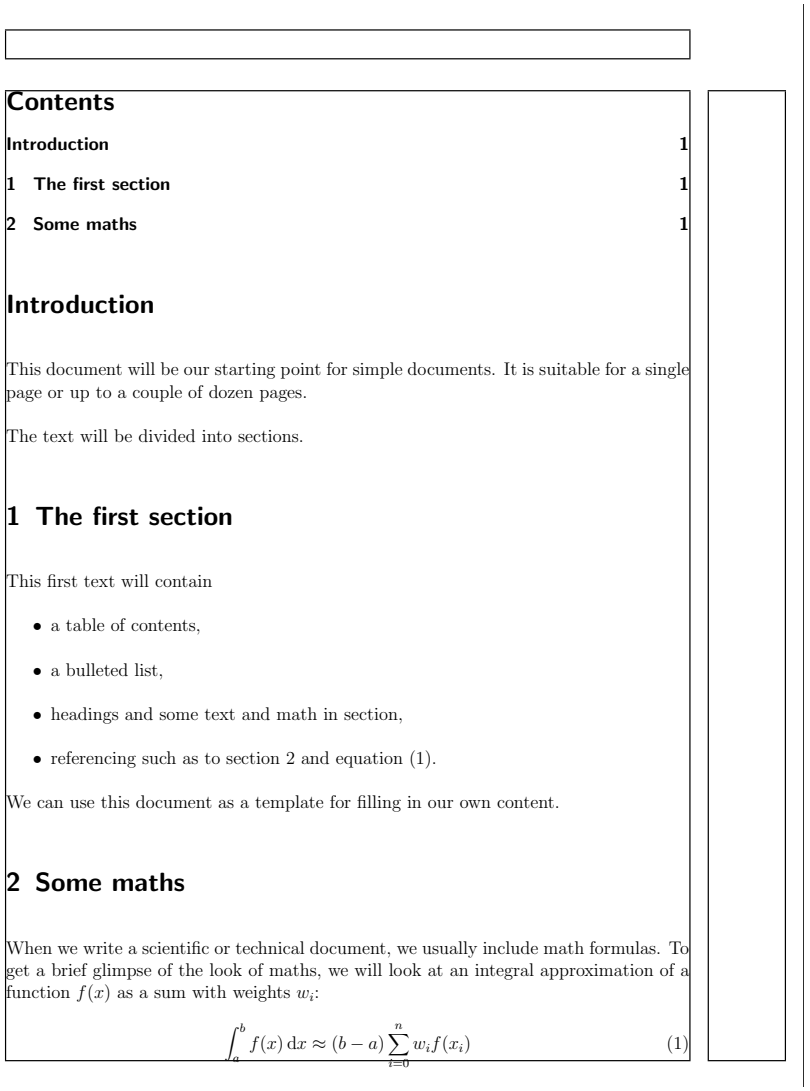
How to do it...

Load the `showframe` package. This means the following steps:

1. Open your document or a sample for testing. Here, we will use the very first document from *Chapter 1*, *The Variety of Document Types*.
2. Add the following line at the end of your preamble:

```
\usepackage{showframe}
```

3. Compile the document. You will see frames around text body, margin note area, and header and footer:



Contents

Introduction 1

1 The first section 1

2 Some maths 1

Introduction

This document will be our starting point for simple documents. It is suitable for a single page or up to a couple of dozen pages.

The text will be divided into sections.

1 The first section

This first text will contain

- a table of contents,
- a bulleted list,
- headings and some text and math in section,
- referencing such as to section 2 and equation (1).

We can use this document as a template for filling in our own content.

2 Some maths

When we write a scientific or technical document, we usually include math formulas. To get a brief glimpse of the look of maths, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \qquad (1)$$

4. Examining the layout can give you ideas about making adjustments, such as changing the margins or other page dimensions. If you don't need these help lines any more, you can disable the package by commenting out `\usepackage{showframe}` or by deleting it.

How it works...

We simply loaded the `showframe` package. It prints the desired help lines on all pages of the document. This package belongs to the `eso-pic` bundle, which will help us in the recipe *Absolute positioning of text* later in the current chapter. As the `showframe` package visualizes layout dimensions, it can help with positioning in relation to margins or text body.

There's more...

There are alternatives to the `showframe` package. Let's take a look at them.

Using geometry

If you have already loaded the `geometry` package to specify page dimensions such as paper size and margin, you could add the option `showframes`, instead of loading the separate package with the same name, in the following ways:

- You can add it while loading the package, as follows:

  ```
  \usepackage[a4paper,bindingoffset=5mm,showframe]{geometry}
  ```

- You can load it by calling the `\geometry` command after the `geometry` package has already been loaded, but also in the preamble:

  ```
  \geometry{showframe}
  ```

Examining the page layout details

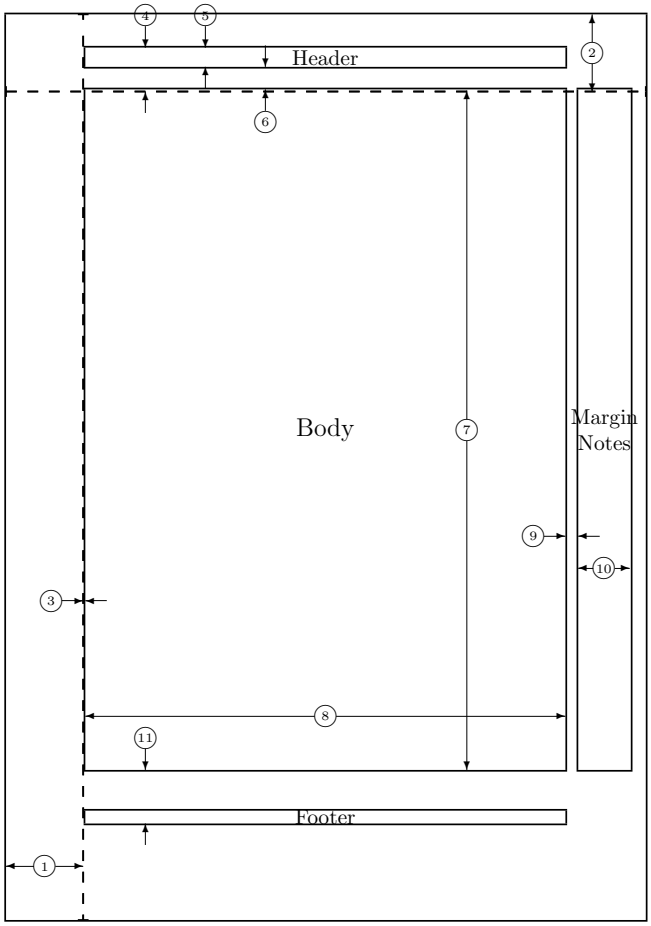
The `layout` package provides a command that prints an overview of the layout, and, in addition, the values of various page layout variables. Let's take a look at the following steps:

1. Load the `layout` package:

   ```
   \usepackage{layout}
   ```

2. Insert the command `\layout` right after the `\begin{document}` command.

3. Take a look at the output, here done with our very first recipe in *Chapter 1*, *The Variety of Document Types*:




```
 1  one inch + \hoffset      2  one inch + \voffset
 3  \oddsidemargin = 2pt     4  \topmargin = -41pt
 5  \headheight = 18pt       6  \headsep = 21pt
 7  \textheight = 635pt      8  \textwidth = 448pt
 9  \marginparsep = 12pt    10  \marginparwidth = 49pt
11  \footskip = 50pt            \marginparpush = 6pt (not shown)
    \hoffset = 0pt              \voffset = 0pt
    \paperwidth = 597pt         \paperheight = 845pt
```

This image is completely generated by the `\layout` command, so it can be seen in the manual and on websites. But it shows the current values of the specific document.

Visualizing boxes of letters and symbols

LaTeX puts text and symbols in boxes and arranges them. There are boxes of whole paragraphs, but also of single letters and symbols. In this recipe, we will closely examine the dimensions of those boxes. We will also take a look at the spacing in between and the resulting dimensions of dynamically adjusted space. This will give us a better understanding of the typesetting.

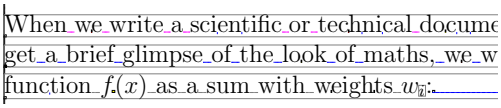
How to do it...

We will additionally load the package `lua-visual-debug`. Then, we will compile the document with **LuaLaTeX**, as follows:

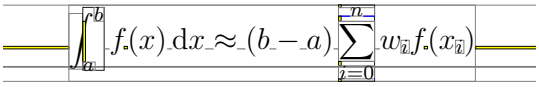
1. Open your document or any sample file. Again, we will use the document from our first recipe in *Chapter 1*, *The Variety of Document Types*.
2. Add the following line at the end of your preamble:

   ```
   \usepackage{lua-visual-debug}
   ```

3. Switch to LuaLaTeX in your editor for typesetting and compile the document. Take a look at these cutouts of the output. The text has some annotations, as follows:



   Formulae consist of many small boxes:



Examining the boxes may give you ideas about possibly tweaking formulas. In *Chapter 10*, *Advanced Mathematics*, you can read about fine-tuning math formulas. If you don't need the box visualization lines any more, you can disable the package by commenting out or deleting the line `\usepackage{lua-visual-debug}`.

How it works...

We simply loaded the `lua-visual-debug` package, which requires LuaLaTeX because of its dependency on LuaTeX. It did all the work for us, and now we just need to understand the output:

- LaTeX's boxes are drawn with thin borders. A zero-width box would be a red rule.

- A filled rectangle means a **kern**. This is a fixed vertical or horizontal space. A positive kern is colored yellow. A negative kern is colored red.
- Tick lines stand for **glue**. This means vertical or horizontal space. In contrast to kern, it can be stretched or shrunken. The lines start and end with a tick. So you can recognize places where glues are touching.
- The blue rectangle below the base line marks a point where hyphenation is allowed.
- A square means a **penalty**. This is an internal value, which TeX tries to minimize in its line-breaking algorithm. A blank square means a maximum penalty; otherwise, it's filled gray.

This visualization is intended to help you with debugging the document's typesetting.

Even if you are using pdfLaTeX, it's often possible to compile by LuaLaTeX for this purpose. Otherwise, you could debug with a simplified copy of the document part you need to debug with LuaLaTeX.

Typesetting in a grid

Besides full horizontal justification, LaTeX also adjusts the page content vertically to get a consistent page height. So, internal spacing is often variable. Furthermore, implicit spaces are often independent of the baseline height.

Consequently, lines of adjacent pages may look shifted. For two-sided prints with very thin paper, matching base lines would look much better. Especially in two-column documents it may be desirable to have baselines of adjacent lines at exactly the same height.

In this recipe, we would like to arrange lines on a grid. Normal text lines will be placed at a baseline grid. Displayed formulas, figures, tables, and captions are allowed to have a different baseline, but the following text should return to the grid.

How to do it...

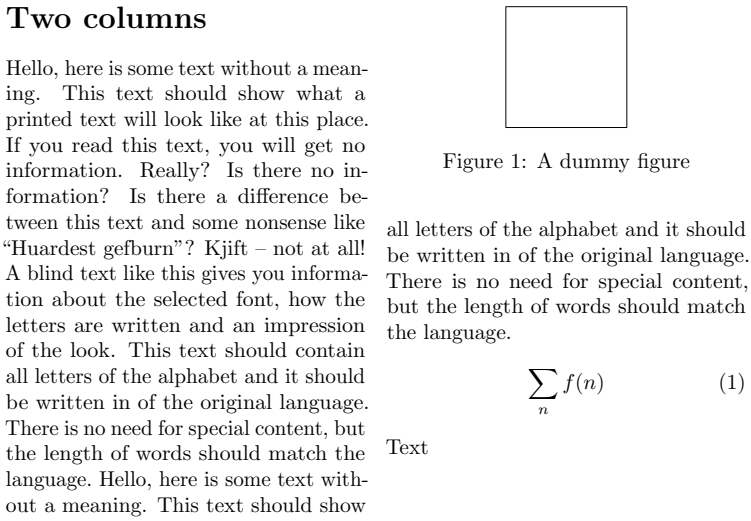
We will use the `grid` package, which has been developed for grid typesetting. Follow these steps:

1. Prepare a small, two-column example with dummy text, where we can apply the `grid` commands. Here is a simple code snippet that you can use for a start:

```
\documentclass{article}
\usepackage[english]{babel}
\usepackage{blindtext}
\usepackage{microtype}
\begin{document}
\twocolumn
```

```
\section*{Two columns}
\blindtext[3]
\begin{figure}
  \centering
  \fbox{\makebox(50,50){}}
  \caption{A dummy figure}
\end{figure}
\begin{equation}
  \sum_n f(n)
\end{equation}
Text
\end{document}
```

2. Take a first look. Compare the height of the text lines in the left- and right-hand columns. Note that the baselines of the text are not yet aligned:



Two columns

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show

Figure 1: A dummy figure

all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$\sum_n f(n) \qquad (1)$$

Text

3. Now, add the `grid` package to the preamble with font size options, as follows:

```
\usepackage[fontsize=10pt,baseline=12pt]{grid}
```

4. Then, put the equation into a `gridenv` environment. It will look like this:

```
\begin{gridenv}
  \begin{equation}
    \sum_n f(n)
  \end{equation}
\end{gridenv}
```

5. Compile the document and take a look at what has changed. Examine the baselines in the following screenshot closely:

Two columns
Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like

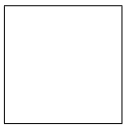


Figure 1: A dummy figure

"Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$\sum_{n} f(n) \qquad (1)$$

Text

How it works...

We specified the font size and the base line height as options to the `grid` package because the default settings don't look right.

The `grid` package makes some changes that helps the lines to stay in the grid. This is how it works:

- So-called glue, also known as **rubber space**, which is elastic space, is removed or replaced by a fixed space.
- The heights of many frequently-used items were made integral multiples of the baseline height so that they would fit exactly to the grid. This has been made, for example, for the following:
  - Section headings
  - Figures and tables
  - Displayed equations



It's recommended that you enclose the `equation` environments within a `gridenv` environment for proper spacing.

We did this here, and got a quick solution for double-column grid typesetting, at least in a limited way. For example, we need to accept the changed section heading size.

There's more...

The `gridset` package provides a command, `\vskipnextgrid`, which jumps the next grid position. It can be used as a simple fix, without being a complete solution. It requires two or more typesetting runs until all is adjusted. So you need to look at the affected place and repeat compiling if needed, especially if you use `\vskipnextgrid` repeatedly.



At the time of writing, the package was in alpha status.

This command can also be used in addition to the `grid` package. In a case where the `grid` package might fail, we can correct this by inserting the `\vskipnextgrid` command.

For example the `amsmath` package's multiline displayed equations could be a problem for the `grid` package. We can fix this as follows:

1. Load `amsmath` using the following command:

   ```
   \usepackage{amsmath}
   ```

2. Also load the `gridset` package by using the following command:

   ```
   \usepackage{gridset}
   ```

3. In the document, create an `align` environment, as follows:

   ```
   \begin{align}
     y &= \sum_{n=1}^3 f(n) \\
       &= f(1) + f(2) + f(3)
   \end{align}
   ```

4. Directly after it, add the following line. It ends the paragraph, skips to the next position in the grid, and avoids inserting paragraph indentations:

   ```
   \par\vskipnextgrid\noindent
   ```

5. Add some text following it.
6. Compile the document once. The position of the previously-added text may still not yet fit the grid.
7. Compile again. The `\vskipnextgrid` command will again adjust the spacing to match the base line grid.

The `\vskipnextgrid` command may have problems in two-column mode. In this case, we fixed it by breaking the paragraph, adjusting the position, and suppressing the following paragraph indentation, because that is unusual after a displayed formula.

Absolute positioning of text

LaTeX takes care of full justification, text height balancing, and placing floating objects such as figures and tables. It does a great job, but sometimes one may need to tell LaTeX to place text or an image exactly at a certain position on a page.

Most positioning commands work in relation to the current position in the document. Now we would like to output text at an absolute position.

How to do it...

We will use the `eso-pic` package for positioning. We will print text at the edge of the page, in the middle, and at specific positions. We will break down the code in to fine steps. However, you can copy the code as a whole from the book's website at `https://www.packtpub.com` or from `http://latex-cookbook.net`.

Follow these steps:

1. Start with any document class. We chose the `article` package with A5 paper here:
   ```
   \documentclass[a5paper]{article}
   ```
2. Load the `lipsum` package so that you can generate dummy text:
   ```
   \usepackage{lipsum}
   ```
3. Load the `graphicx` package; we will later use its rotating feature:
   ```
   \usepackage{graphicx}
   ```
4. Load the `showframe` package for visualizing page dimensions, just to help us in the draft stadium:
   ```
   \usepackage{showframe}
   ```
5. Load the `eso-pic` package, which does the positioning for us:
   ```
   \usepackage{eso-pic}
   ```
6. Load the classic `picture` package to place with coordinates:
   ```
   \usepackage{picture}
   ```

7. We will print the page numbers by ourselves, so disable the original page numbering:

```
\pagestyle{empty}
```

8. Start the document:

```
\begin{document}
```

9. Add the page number to all pages as follows:

```
\AddToShipoutPictureBG{%
  \setlength{\unitlength}{1cm}%
  \put(2.5,2){Test document}%
  \put(\paperwidth-2cm,2cm){\llap{\thepage}}%
}
```

10. Add some text to a single page as follows. Note the star (`*`):

```
\AddToShipoutPictureBG*{%
  \AtPageLowerLeft{Page bottom left}%
  \AtPageUpperLeft{\raisebox{-\height}{Page top left}}%
  \AtTextUpperLeft{\raisebox{-\height}{%
    \color{red}Text area top left}}%
}
```

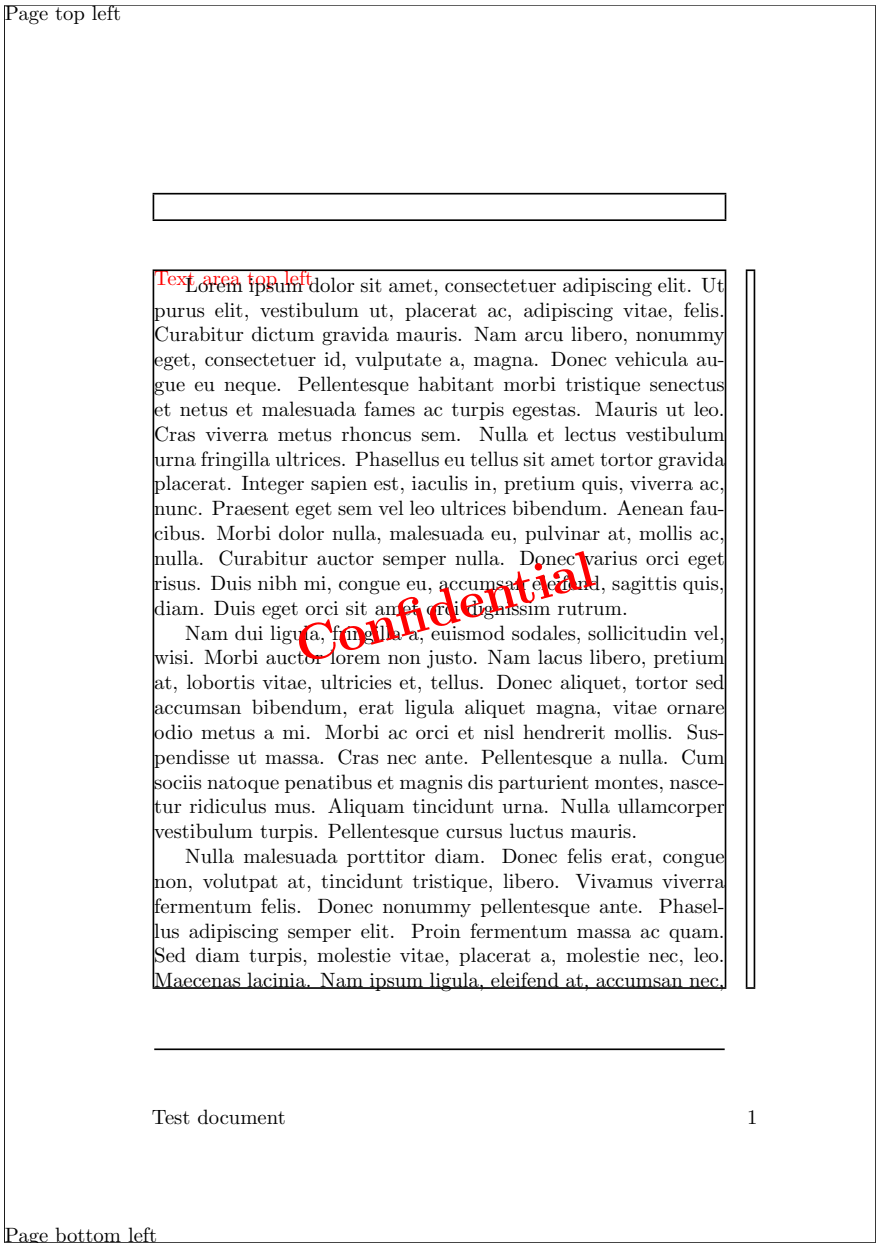
11. Add a confidential sign at the top of the page:

```
\AddToShipoutPictureFG{%
  \AtPageCenter{\rotatebox{15}{\makebox[0pt]{%
    \Huge\bfseries\color{red}Confidential}}}%
}
```

12. Print some dummy text. It starts at the beginning of the page, as usual. Then, end the document:

```
\lipsum
\end{document}
```

13. Compile the document and take a look at the first page:



How it works...

We loaded several packages, as follows:

- The `lipsum` package gives us dummy text, which we get by the command `\lipsum`.
- The `graphicx` package is for including images, but here we loaded it because we use its command `\rotatebox` to rotate text in step 11.
- The `showframe` package prints lines around text and margin areas, as you saw in a recipe before.
- The `eso-pic` package does the main job here. It provides commands for printing text or graphics independent of the current position on the page. It can do this in the background, which means behind text, but also in the foreground, overwriting the normal text.
- The `picture` package is a helper package. Some commands, such as `\put`, expect arguments with simple numbers, which are interpreted as factors of `\unitlength`. This package allows the use of different lengths with arbitrary units.

We set an empty page style; that is, one without header and footer text. In the document, we called the `eso-pic` commands:

- The `\AddToShipoutPictureBG` command takes LaTeX's `picture` commands, which are put into a zero-length `picture` environment in the lower-left corner of the page. This is printed onto the page background or behind the normal text.

  Here, we first defined the base unit length to be 1 centimeter. In the following `\put` command, we used multiples of the base length to print "Test document" near the lower-left corner. Finally, we printed the current page number near the lower-right corner. We calculated with paper width and centimeter values. This syntax is brought by the picture package. The `\llap` command puts its argument right-aligned into a zero-length box.

  This will be repeated for the following pages.

- The `\AddToShipoutPictureBG*` command works like the `\AddToShipoutPictureBG` command, but only on the current page. With the `\AtPageLowerLeft` command, we placed text in the lower-left corner of the page. The `\AtPageUpperLeft` command works similarly for the upper left. We just lowered the text by its own height. The `\AtTextUpperLeft` command is the analogous command for the text area.
- The `\AddToShipoutPictureFG` page works like the `\AddToShipoutPictureBG` command, but for the page foreground. We placed the word "Confidential" in huge red lettering, rotated above the text, in the middle of the page.
- The `\AddToShipoutPictureFG*` command is the analogous command for the foreground, which affects only the current page.

There's more...

There are alternative ways to achieve what we just did. These are packages that do a similar job:

- The `atbegshi` package can be used directly. The `eso-pic` package actually builds on it.
- The `everyshi` package does a similar job. The `atbegshi` package is a modern reimplementation of it.
- The `textpos` package bases on the `everyshi` package. It provides a convenient user interface.

Capable graphics packages provide their own means in addition.

- The package `TikZ` can be used to place text in relation to the `current page` node in `overlay` mode
- The `PSTricks` bundle contains the package `pst-abspos` for putting an object at an arbitrary position on the page

If any of it fits better to your work tools, you may take a closer look. As you saw in other recipes, you can access each package's documentation by calling the `texdoc` command at the command prompt, followed by the package name and return key. For online access, just type the package name into the search field at `http://texdoc.net`.

Starting a paragraph with an over-sized letter

In older texts, such as in books of fairy tales, we sometimes see the first paragraph in a text starting with a huge letter, while the following text flows around it. This is called a **drop cap** or an **initial**. We will now use this design for our own text.

How to do it...

We will use the `lettrine` package, which provides a command for this purpose.
Follow these steps:

1. Start a document with any document class. Here, we chose the `book` class. We will use A6 paper size, simply because this makes the recipe easy to show with little text:

```
\documentclass{book}
\usepackage[a6paper]{geometry}
```

2. Load the `lettrine` package:

```
\usepackage{lettrine}
```

3. Begin the document:

```
\begin{document}
```

4. Start a paragraph with the command `\lettrine{letter}{further introduction}`, as follows:

```
\lettrine{O}{nce upon a time}, professional writer used
a mechanical machine called a typewriter. It commonly
printed fixed-width characters. Emphasizing was done by
writing all capitals, and by underlining.
```

5. End the document:

```
\end{document}
```

6. Compile the document. Now look at the shape of our paragraph:

ONCE UPON A TIME, professional writer used a mechanical machine called a typewriter. It commonly printed fixed-width characters. Emphasizing was done by writing all capitals, and by underlining.

How it works...

For this simple example, we used the basic `book` class. We loaded the `geometry` package to get a handy A6 paper size. The final command in the preamble loaded the `lettrine` package, which provides exactly the design we were looking for. The command `\lettrine{O}{nce upon a time}` prints one big letter `O`, followed by the text in the second pair of braces, which is printed in small caps. The other text flows around the large letter.

There's more...

The design of the dropped capitals can be customized. Let's take a look at some options.

Changing the drop cap size

By default, the dropped capital will cover two lines. You can change the number of lines by setting the optional argument lines as follows:

```
\lettrine[lines=3]{O}{nce upon a time}
```

Now a huge `O` covers three lines. Furthermore, you can enlarge it by setting the `loversize` option, standalone or in combination with the previously seen option:

```
\lettrine[lines=3,loversize=0.2]{O}{nce upon a time}
```

The `loversize` option can be set to a value larger than -1 and smaller than or equal to 1, and means the resize factor. That is, a value of 0.1 means enlarging by 10 percent.

The `lettrine` package provides a `key=value` interface. More options are available to control further aspects, such as the gap between the drop cap and the following text, and vertical shifting. You can also let drop caps hang into the margin. These features are explained in the manual. You can open it using `texdoc lettrine` in Command prompt, or at `http://texdoc.net/pkg/lettrine`.



In case of repeated use:

For making later adjustments easier and consistent, it's recommended that you create a macro via the `\newcommand` command, using the `\lettrine` command inside. It's better than repeatedly typing the `\lettrine` commands with options.

Coloring the initial

The simplest way of getting colored drop caps is by using the well-known commands of the `color` or `xcolor` package, as follows:

```
\usepackage{xcolor}
...
\lettrine{\textcolor{red}{A}}{nother} time
```



A full list of available colors and names is contained in the `xcolor` manual. You can open it by typing the `texdoc xcolor` command in Command Prompt, or you can find it online at `http://texdoc.net/pkg/xcolor`. You may need to set an option to access certain predefined names, such as `svgnames`, `dvipsnames`, or `x11names`.

The package `coloredlettrine` provides an even fancier way. It provides bicolor initials based on the **EB Garamond** font. Internally, the initials are split into two fonts. One provides the background ornaments, the other the actual letters. This allows separate coloring. We will use the **OpenType** version of the EB Garamond font. OpenType requires compilation with XeLaTeX or LuaLaTeX. Many LaTeX editors support them too.

At the time of writing, the package was still in development. There was just a small set of initials available. That's why we will refer to the development sources. Let's take a look at the following steps:

1. Download the package files from `https://github.com/raphink/coloredlettrine`. There should be a `.ins` file and a `.dtx` file. Put them together into the same folder. Compile the file `coloredlettrine.ins` with LaTeX. This produces a file named `coloredlettrine.sty`, which you can place in your TeX installation or your document folder.

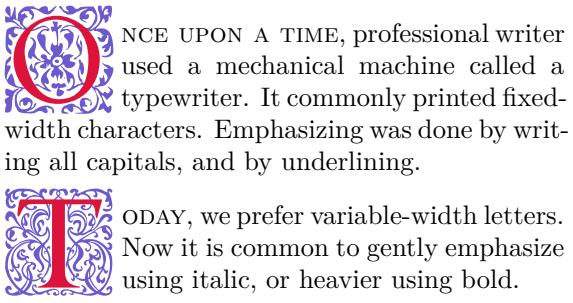
2. Get the latest version of the EB Garamond font from `https://bitbucket.org/georgd/eb-garamond/downloads` and install it. Specifically, unzip the downloaded file and install at least the `EBGaramond-InitialsF1.otf` and `EBGaramond-InitialsF2.otf` software. On a Mac, double click the file to see the contents and then click the install button shown.

3. The package `coloredlettrine` contains an example that you can compile. Here, let's modify our example from the recipe's start:

```
\documentclass{book}
\usepackage[a6paper]{geometry}
\usepackage{coloredlettrine}
\renewcommand{\EBLettrineBackColor}{SlateBlue}
\setcounter{DefaultLines}{3}
\renewcommand{\DefaultLraise}{0.3}
\renewcommand{\DefaultFindent}{0.3em}
\renewcommand{\DefaultNindent}{0pt}
\begin{document}
\coloredlettrine{O}{nce upon a time}, professional
writer used a mechanical machine called a typewriter.
It commonly printed fixed-width characters. Emphasizing
was done by writing all capitals, and by underlining.

\coloredlettrine{T}{oday}, we prefer variable-width
letters. Now it is common to gently emphasize using italic,
or heavier using bold.
\end{document}
```

4. Choose XeLaTeX for typesetting, compile, and have a look:



The command `\coloredlettrine` can be used exactly like the `\lettrine` command since it is just a wrapper for it. You can redefine the macros `\EBLettrineBackColor` and `\EBLettrineFrontColor` to choose the color.

In this example, we set default values for `lettrine` parameters, which we would otherwise have needed to provide as `key=value` options, as explained earlier in this recipe. This way can save you from defining your own macro. It is described in the manual, which you can open with the `texdoc lettrine` command at the command prompt, or find online at `http://texdoc.net/pkg/lettrine`.

Fitting text to a shape

There are occasions when text is not strictly arranged in a rectangular box. For example, if you would like to print a label for a DVD or compact disc, the text should be arranged in a circle.

How to do it...

The `shapepar` package can typeset paragraphs in a specific shape, such as a circle, a hexagon, or a heart. The shape size will be adjusted so the given text fits in. We will now try it with a heart:

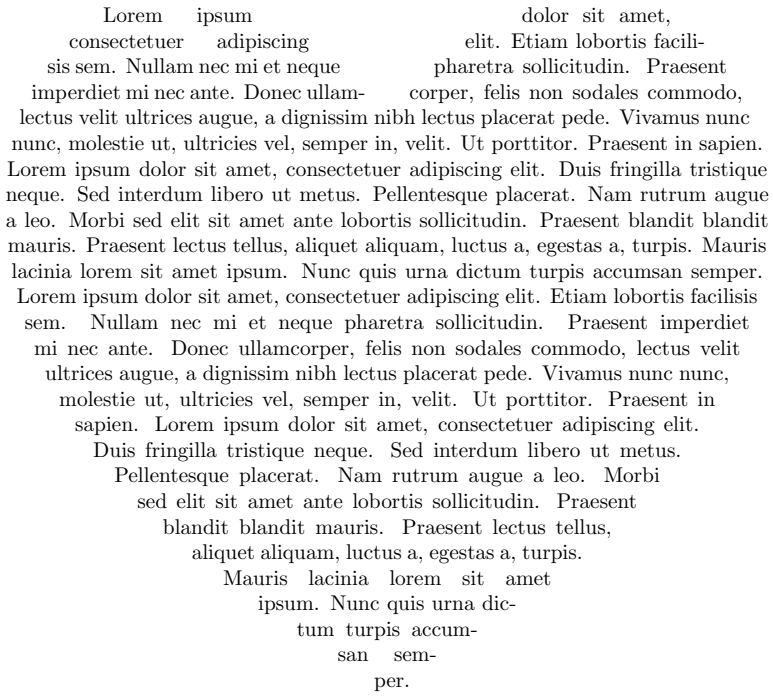
1. Make a small document, load the packages `blindtext` (for dummy text) and the `shapepar` package:

```
\documentclass{article}
\usepackage{blindtext}
\usepackage{shapepar}
```

2. In the document, use the `\shapepar` command with the shape argument, and then text as follows:

```
\begin{document}
\shapepar{\heartshape}\blindtext[2]
\end{document}
```

3. Compile and have a look:



How it works...

We loaded the `blindtext` package, which provides filler text via the `\blindtext` command, which is great for testing. Then we loaded the `shapepar` package.

That package provides the command `\shapepar`, which is used as follows:

```
\shapepar[scale length]{shape command} text of the paragraph
```

Here, we used the `\heartshape` command as the shape command, which typesets the following paragraph of text with the shape of a heart. The shape applies only to the following paragraph. The paragraph is not an argument.

The `scale length` value is optional. It's a LaTeX length, which will be used as a base unit within the shape definition. If not given, it's automatically calculated for optimal filling of the shape.

There's another short command:

```
\heartpar{text}
```

It works as follows:

```
\shapepar{\heartshape} text\ \ $\heartsuit$\par
```

It prints the text in the shape of a heart and ends it with a heart symbol.

There's more...

We can shape text in various ways. Furthermore, we can let text flow around a shape. In other words, we can cut out shapes from the text.

Further shapes

The following shapes and corresponding commands are predefined:

Shape macro	Command	Effect
`\squareshape`	`\squarepar`	Square
`\circleshape`	`\circlepar`	Circle
`\CDshape`	`\CDlabel`	Circle with hole for DVD or compact disk
`\diamondshape`	`\diamondpar`	Diamond (rhomboid)
`\heartshape`	`\heartpar`	Heart
`\starshape`	`\starpar`	Five-pointed star
`\hexagonshape`	`\hexagonpar`	Hexagon
`\nutshape`	`\nutpar`	Hexagon with hole
`\rectangleshape{height}{width}`		Rectangle

The last shape doesn't provide another command. Use it as follows:

```
\shapepar{\rectangleshape{40}{20}} text
```

Use it without the units for the length, as the command refers to the base unit length of the shape. The holes mentioned in the preceding table are circular.

Cutting out shapes

A companion of the `\shapepar` command can cut out text using a shape:

```
\cutout {side} (horizontal offset,vertical offset)
  settings \shapepar ...
```

This cuts out the text with the specified shape from the following text. The `side` option can be `l` or `r` for left or right, respectively. You can use offsets for shifting settings as optional code. It can contain commands such as modifying the `\cutoutsep` command, which is the separation between outer text and shaped text, 12 pt by default. These commands have only a local effect.

For further details, refer to the package's manual. Since a good example is missing there, here's one, simply working within the code from our preceding recipes:

```
\cutout{l} (5ex,2\baselineskip) \setlength{\cutoutsep}{8pt}
  \shapepar{\circleshape} a few words of text\par
\blindtext
```

This will generate the following as output:

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent a few imperdiet mi nec ante. Donec ullamcorper, felis non sodales comwords of modo, lectus velit ultrices augue, a dignissim nibh lectus placerat text pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Creating a pull quote

To entice people to read a text, we can present a short, attractive excerpt as a quotation. That means we pull out some text. In a two-column layout, it's looks nice to put the quotation into a window in the middle of the page between the two columns, with the regular text flowing around it. It's also a nice way of embedding images.

How to do it...

One possibility is to use the `shapepar` package to cut out space from the text, like in the previous recipe. However, it would be a bit challenging doing it twice, once for each column.

The `pullquote` package provides a solution. It can typeset a balanced two-column text layout with a cut-out window. This can be filled with text or an image. The shape is arbitrary.

We will use dummy text and highlight a quotation from Donald Knuth, the creator of TeX:

1. Download the `pullquote.dtx` file from `http://bazaar.launchpad.net/~tex-sx/tex-sx/development/view/head:/pullquote.dtx` or from CTAN, once it's provided there too.

2. Click on **browse files** and also download the `pq-alice.jpg`, `pq-duck.pdf`, and `pullquote_test.tex` files, to get filler images and a sample file.
3. Compile the `pullquote.dtx` file with LaTeX. Do it twice for correct references. It will generate a `pullquote.sty` file and the documentation file `pullquote.pdf`. Place the files where your LaTeX can find them, or simply in the document's folder.
4. Start with a document class and the `lipsum` package for dummy text, and load the `pullquote` package:

```
\documentclass{article}
\usepackage{lipsum}
\usepackage{pullquote}
```

5. Create a command that prints your quotation in a paragraph box:

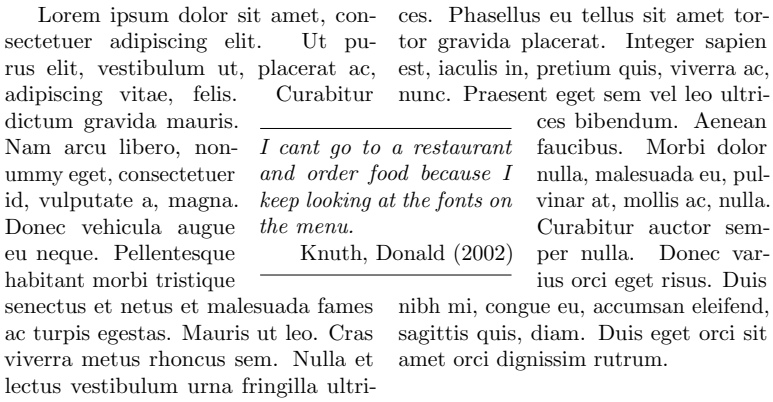
```
\newcommand{\myquote}{%
  \parbox{4cm}{
    \hrule\vspace{1ex}
    \textit{I can't go to a restaurant and order food
      because I keep looking at the fonts on the menu.}

    \hfill Knuth, Donald (2002)%
    \vspace{1ex}
    \hrule
  }%
}
```

6. In the document, use a `pullquote` environment with the self-defined macro in the argument, and dummy text, as follows:

```
\begin{document}
\begin{pullquote}{object=\myquote}
  \lipsum[1]
\end{pullquote}
\end{document}
```

7. Compile and have a look:



How it works...

We loaded the `lipsum` package, which gives us filler text. Then we loaded the `pullquote` package. We defined a macro, which prints the quotation.

For our example, we made a paragraph box with lines above and below, printing italic text and the author's name at the right. To avoid undesired white space, we commented out the line breaks in some places by putting a `%` sign at the end.

In the document, we simply made a `pullquote` environment with that macro as a pulled object within the argument, and `lipsum` text for filling the surrounding space.

The `pullquote` package does the rest for us. It does the following things:

- It puts the object into a box
- It measures height and width
- It adds space for the distance
- It normalizes the total height to be an integral multiple of the value specified for the `\baselineskip` command to match a number of text lines
- It calculates the vertical position
- It calculates a shape for the paragraph
- It balances the text columns according to the shape
- It arranges all and prints the whole construct

There are some restrictions. Mainly, the text within the environment should be simple paragraphs of text. This means that lists such as itemize, displayed math, section headings, and modified vertical spacing in general, are undesirable. Such non-simple elements may trouble the calculation. However, it's already great for images and text boxes.

There's more...

The `key=value` list in the argument of the `pullquote` package understands further parameters. Besides the default rectangular shape, there's a circular one. We can specify it via the `shape` option. The following example will demonstrate it. We will create a TikZ picture to have an actual circled element to place in. So, we need to load the TikZ package as well.

You can verify it with this sample code, similar to the preceding code:

```
\documentclass{article}
\usepackage{lipsum}
\usepackage{pullquote}
\usepackage{tikz}
\newcommand{\mylogo}{%
  \begin{tikzpicture}
    \node[shape=circle,draw=gray!40,line width=3pt,
      fill={gray!15},font=\Huge] {\TeX};
  \end{tikzpicture}%
}
\begin{document}
\begin{pullquote}{shape=circular,object=\mylogo}
  \lipsum[1]
\end{pullquote}
\end{document}
```

The code will produce this layout:

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.



TikZ is a very capable graphics package. You can read more about it in *Chapter 9*, *Creating Graphics*. We could also have used a circular image instead. For this, refer to the recipe *Shaping an image like a circle* in the next chapter.

For arbitrary shapes specified by shape functions, refer to the package manual. It also explains image inclusion support with the `shape=image` list. Then, the cut-out dimensions would be calculated from the image dimensions. For this, the free **ImageMagick** program would need to be installed on the system, since it would then be called internally.

3
Adjusting Fonts

This chapter contains recipes that will help you to benefit from the wealth of fonts. We will specifically cover the following:

- Choosing a document font
- Locally switching to a different font
- Importing just a single symbol of a font
- Writing bold mathematical symbols
- Getting the sans serif mathematics font
- Writing double stroke letters as if on a blackboard
- Enabling the searching and copying of ligatures
- Suppressing ligatures
- Adding a contour

Introduction

When LaTeX was young, documents looked quite similar, since there was a precious little choice of fonts. Over time, many new fonts were invented and gained support from LaTeX.

To find the perfect fonts for your documents, visit the LaTeX font catalogue at `http://www.tug.dk/FontCatalogue/`.

In this chapter, we will see how to choose fonts globally and how to adjust them within the document.

Before we take off, let's take a quick look at basic LaTeX commands for switching between fonts. We keep it very short, as introductory texts usually cover it. Experienced LaTeX users may skip this section.

Basic font commands

Fonts for text have five main attributes:

- **Encoding**: We covered this in the previous chapter and concluded that T1 encoding is usually a good choice for common Latin text, which can be activated by the following command:

  ```
  \usepackage[T1]{fontenc}
  ```

- **Family**: This is what we we call sets of fonts of the same origin and the same type. You can switch to a font family using one of these commands:
  - `\rmfamily`: This is the default that switches to Roman font, which means text with serifs.
  - `\sffamily`: This switches to sans serif font.
  - `\ttfamily`: This is for typewriter font.
- **Series**: This denotes how heavy a font is. Following are the commands:
  - `\bfseries`: This is for bold font.
  - `\mdseries`: This is for medium font, which is the default.
- **Shape**: This attribute relates to the shape of the text; a font family can have several shapes:
  - `\upshape`: This switches to upright text, which is the default
  - `\itshape`: This switches to italics.
  - `\slshape`: This gives slanted text.
  - `\scshape`: This is for small capitals.
- **Size**: The base size is specified as an option to the document class, as we did earlier. We can choose a different size:
  - We get a larger text size in the increasing order using `\large`, `\Large`, `\LARGE`, `\huge`, and `\Huge`.
  - We get lower text size using `\small`, `\footnotesize`, `\scriptsize`, and `\tiny`.
  - We can switch back to the normal size using `\normalfont`.

The effect of these commands is limited by environments. You can also limit the effect by curly braces, for example, {\bfseries ... }.

There's another more consistent syntax for modifying short pieces of text. For example. the following commands choose a font family for a piece of text:

- \textrm{...} switches the argument to Roman text, \textsf{...} switches it to sans-serif, and \texttt{...} switches the argument to typewriter text.

Similarly, the following applies to the series and shape:

- \textbf{...} makes the argument bold, \textmd{...} makes it medium.
- \textit{...} applies *Italics*, \textsl{...} gives slanted text, \textsc{...} switches the argument to small capitals, and \textup{...} is for upright font.

That's easier to remember. There's no such command for font size. It's unusual anyway to manually change the font size within surrounding text. Just think of calculating the interline distance for the paragraph.

There's more...

Classes and packages use these font commands to define consistent styles. As the author, we state elements such as section headings, footnotes, subscripts, and emphasized snippets; while LaTeX chooses the corresponding size, shape, and series.

What if you would like to use those commands yourself? It's a very good practice to use such font commands only in the preamble, in macro definitions, not in the document's body text. For example, instead of scattering the \textbf{...} commands all over the text for bold author names, you should define an author name style using:

```
\newcommand{\authorname}[1]{\textbf{#1}}
```

This allows to implement easy and consistent changes at a single place—the preamble, for example, when you would decide to use small caps or *Italic*s instead.

A macro for each required formatting brings logic into the text. And you can modify all occurrences at the same time by changing the macro.

Choosing a font for a document

The default font has the name **Computer Modern** and is of very good quality. It is actually a whole font family that contains bold, italic, sans-serif, typewriter, and more font versions. All the fonts are well composed to fit together.

If you would like to change the font, use a complete bundle or carefully select font families based on shape and size. That's because, besides giving a good appearance to the document overall, it is important that all font families are compliant to each other when they are used together.

In this recipe, we will take a look at some font sets and recommended combinations.

Getting ready

For the fonts you would like to use, their files should already be installed on your TeX system. If not already present, install them. Use the package manager of your LaTeX distribution, such as the `MiKTeX` package manager in case of MiKTeX on Windows, or the TeX Live manager tool called `tlmgr`.

In case you have TeX Live installed, it offers the possibility to install entire font collections, so you may run the `tlmgr` tool at the command line:

```
tlmgr install collection-fontsrecommended
tlmgr install collection-fontsextra
```

If sufficient hard disk space is available, installing all fonts or even all packages of the TeX distribution can save you from some headache later, so you would not miss any fonts.

TeX Live installs only free fonts that have no restrictions on distribution by the supplier. There's a tool called `getnonfreefonts` for downloading and installing other fonts. For documentation and download, visit `http://www.tug.org/fonts/getnonfreefonts/`.

How to do it...

Fonts with LaTeX support often come with a package. As the author, you can load the package and that package takes care of loading and activating fonts.

So, commonly these steps would be followed in the preamble:

1. Switch to the required font encoding. Most modern fonts work with T1, as explained in *Chapter 2*, *Tuning the Text*:

   ```
   \usepackage[T1]{fontenc}
   ```

2. Load the font package:

   ```
   \usepackage{fontname}
   ```

   The `fontname` parameter is the name or the short form for the name of a font that you need to know, for example, by visiting the LaTeX Font Catalogue.

3. If your default document font should be sans-serif, you can switch to that:

   ```
   \renewcommand{\familydefault}{\sfdefault}
   ```

In the next recipe, you will see another method of choosing a font, which also works locally. Here, we continue with the document wide font choice.

There's more...

Let's take a look at some good alternative fonts. Use each code given next together with the setting `\usepackage[T1]{fontenc}`.

We will take a look at a sample output so that you can compare the font selections. Note that you may see a lower quality than in a LaTeX document, as the publisher of this book accepts only images in bitmap format. This can result in a blurry or pixelated output, depending on the print or screen resolution.

You can also see the following example with the PDF output at `http://latex-cookbook.net/cookbook/examples/tag/fonts/`.

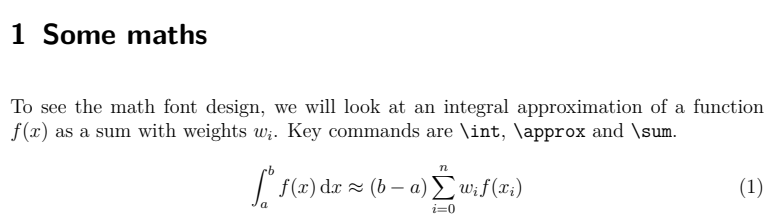
To demonstrate this, we will take our very first example in the first chapter, just a bit reduced. We will see how roman, sans serif, italic, and typewriter fonts harmonize.

Latin Modern

Latin Modern is very similar to the default Computer Modern font. It is of excellent quality. This font bundle covers the usual requirements, such as having a serif, sans serif, typewriter, and symbol font. Load the fonts using the following command:

```
\usepackage{lmodern}
```

Latin Modern gives us the following appearance:



1 Some maths

To see the math font design, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i. Key commands are `\int`, `\approx` and `\sum`.

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \qquad (1)$$

It is still similar to the one in our very first recipe.

Kepler fonts

The **Kepler fonts** are a complete and well-designed set of fonts in various shapes. There are upright and slanted Greek letters, bold math glyphs, old style numbers, and several weights from light to bold extended. Even slanted small caps are available. You can get the whole set using this command:

```
\usepackage{kpfonts}
```

The appearance of our sample changes to the following:

1 Some maths

To see the math font design, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i. Key commands are `\int`, `\approx` and `\sum`.

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \tag{1}$$

The typefaces harmonize well. They are not very heavy, for example, the typewriter code better matches the greyness of the normal text, since it doesn't look as heavy as it did in the previous sample.

Font combinations

Some fonts exist only as text fonts, math fonts, or in a specific shape. In such a case, it is a challenge to combine different fonts, both regarding taste and scaling. Here are some examples that work fine together:

```
\usepackage{libertine}
\usepackage[libertine,cmintegrals,cmbraces,vvarbb]{newtxmath}
\usepackage[scaled=0.95]{inconsolata}
```

Together they give us this appearance:

1 Some maths

To see the math font design, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i. Key commands are `\int`, `\approx` and `\sum`.

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \tag{1}$$

We loaded the **Linux Libertine** font. It is not as dense as a **Times** font, but denser than Computer Modern. Then, we added the `newtxmath` font, but programmed it to use **Libertine** glyphs when appropriate other than using the Computer Modern integral and braces symbols. The last option tells `newtxmath` to use another good blackboard font with `\mathbb`. Try this selection and check whether you like it, omit options to compare with the default settings. For details and further features, refer to the `newtx` manual via `texdoc`.

Finally, we loaded the excellent monospaced font **Inconsolata** for typewriter font-like shape, and scaled it a bit to better match our text font.

Now, let's try another combination:

```
\usepackage[sc,osf]{mathpazo}
\usepackage[T1,small,euler-digits]{eulervm}
\usepackage[scaled=0.86]{berasans}
\usepackage[scaled=0.84]{beramono}
```

You can see the difference, such as more upright shape for variables and integral and summation signs:

1 Some maths

To see the math font design, we will look at an integral approximation of a function $f(x)$ as a sum with weights w_i. Key commands are `\int`, `\approx` and `\sum`.

$$\int_a^b f(x)\,dx \approx (b-a)\sum_{i=0}^{n} w_i f(x_i) \qquad (1)$$

We loaded the `mathpazo` package, which gives us a **Palatino** text font. We replaced its Palatino Italic-like math font by the **Euler Virtual Math** font. It's basically the Euler font with missing symbols taken from Computer Modern, which is why it is called *virtual*.

In addition, we loaded the **Bera** sans serif font and the monospaced Bera shape for the typewriter text; both are appropriately scaled.

You may have noticed the options `sc` and `osf` for `mathpazo`, that is, for Palatino, which gives us a small caps font and uses old style figures as the default.

Now you have some good suggestions for a fine font selection. Using any of them, your document will look professionally designed and still different from a LaTeX standard document.

Locally switching to a different font

A typographically good document with consistent appearance commonly uses just a few fonts, each one with a purpose. Common font choices are:

- Serif body text
- Sans serif for headings
- Monospaced for source code

Each font family is defined in the preamble, usually implicitly done by packages. There are LaTeX commands for switching between families, shapes, and weight. But what if you would like to additionally use a completely different font, such as a second serif font? This recipe will help to achieve that.

How to do it...

We will take a look at two ways:

- Defining a command
- Defining an environment

In each case, we need to know the *code* for the font. Such code is based on the Karl Berry's naming scheme; you can read about it by typing the `texdoc fontname` command in the Command Prompt, or at `http://texdoc.net/pkg/fontname`. You don't really need to study this guide, just take a look at the font's own documentation. For your convenience, here is a selection of frequently-used font families and their codes:

Font family name	Font family code
Avant Garde	pag
Bookman	pbk
Charter	bch
Computer Concrete	ccr
Computer Modern Roman	cmr
Computer Modern Sans Serif	cmss
Computer Modern Typewriter	cmtt
Courier	pcr
Garamond	mdugm

Font family name	Font family code
Helvetica	phv
Inconsolata	fi4
Latin Modern	lmr
Latin Modern Sans Serif	lmss
Latin Modern Typewriter	lmtt
New Century Schoolbook	pnc
Palatino	ppl
Times	ptm
Utopia	put
Zapf Chancery	pzc

We will use the last one for our recipe.

A command for changing the font

To avoid repetition of command sequences within the document, we will define a macro for changing the font. This allows consistent future adjustments.

Define a simple macro for the change to the **Zapf Chancery** font. Write it into your preamble:

```
\newcommand{\zapf}{\fontfamily{pzc}\selectfont}
```

In your document, use it to switch the font. Group the command with the text to be affected. Consider ending the paragraph before you end the group, which can be done by inserting an empty line or by the `\par` command.

You can group the command by using curly braces:

```
{\zapf Text in Zapf Chancery\par}
```

Here is another way, which may be a better choice for you to understand clearly:

```
\begingroup
\zapf
Text in Zapf Chancery
\par
\endgroup
```

An environment for changing the font

Creating an environment is an even clearer way of restricting the effect of the change. Write it into your preamble:

```
\newenvironment{zapfenv}{\fontfamily{pzc}\selectfont}{}
```

Now, you can use this environment in your document:

```
\begin{zapfenv}
Text in Zapf Chancery
\par
\end{zapfenv}
```

You can save some work by moving the final paragraph break to the environment definition. This way, you don't always need to type before closing the environment. The definition will be as follows:

```
\newenvironment{zapfenv}{\fontfamily{pzc}\selectfont}{\par}
```

How it works...

At first, we choose the font family, which has not yet had an immediate effect on the font. We need to call the `\selectfont` command in order to apply the change.

Both grouping and the use of an environment keep the change local. This means that after ending the group or environment, the following text will have the same font as earlier.

The font properties at the end of the paragraph determine how TeX formats a paragraph, especially its line spacing. That's the reason for having the paragraph break *within* the group or environment, not directly afterwards. Otherwise, for example, in case of switching to a bigger font, we can have a paragraph with a big font but with a small line spacing from the outer font.

There's more...

If you would really like to use several different font families within a document, you can make the macro and the environment more variable by introducing an argument. The definitions can be changed to:

```
\newcommand{\setfont}[1]{\fontfamily{#1}\selectfont}
\newenvironment{fontenv}[1]{\fontfamily{#1}\selectfont}{\par }
```

Also, the usage of the command will change to:

```
{\setfont{pzc} Text in Zapf Chancery\par}
```

Alternatively, it changes to this:

```
\begin{fontenv}{pzc}
Text in Zapf Chancery
\end{fontenv}
```

There's also a command for switching back to the default font family. You can use it to explicitly reset the default font family of the document:

```
\normalfont
```

You can change more parameters of the chosen font. In such cases, you may define some macros for several purposes, such as various heading fonts. Look at this sample:

```
\newcommand{\latin}{\fontencoding{T1}\fontfamily{lmr}%
  \fontshape{sl}\fontseries{b}\fontsize{16pt}{20pt}\selectfont}
```

Here, we switch to T1 encoded Latin Modern Roman with slanted shape and bold weight. Furthermore, we set a font size of 16 pt and a line spacing of 20 pt. For detailed information, refer to the Guide *LaTeX 2e font selection*, which is accessible on the command line by typing the `texdoc fntguide` command in the Command Prompt. It is available for download at `http://texdoc.net/pkg/fntguide`.

As with all physical font settings, such changes should not be directly called within a document. They are useful within global formatting commands, for example, to use the Zapf Chancery font in KOMA-Script chapter headings:

```
\setkomafont{chapter}{\normalcolor\zapf\Huge}
```

Generally, once you know how to adjust the font, you can apply these macros within the *logical* macros, such as for defining the appearance of keywords, code, or hyperlinks.

Importing just a single symbol from a font family

There are a lot of packages providing symbols. Often, you get the new commands for additional symbols by simply loading the package by using the `\usepackage` command. However, there can be name conflicts in case the other packages already use the same command name. It can result in an error or in silently overwriting the command.

In this recipe, we will see how to choose one or more specific symbols from a package and access them without loading the whole package.

We will choose a binary relation symbol from the `mathabx` package. This will be the sign for *less or equal*. Later, we will import its negation.

Getting ready

In this recipe, we need to take a look at the source code of the symbol package to imitate part of what it does, so prepare yourself:

1. Locate the file `mathabx.sty` and open it. At the Command Prompt, `kpsewhich mathabx.sty` gives you the location. But you can also use your file manager.
2. In `mathabx.sty`, you can see `\input mathabx.dcl`. There are symbol declarations; open this file as well. It is in the same folder, but again `kpsewhich mathabx.dcl` will find it.
3. Type `texdoc fntguide` to open the Guide to LaTeX 2e font selection. This is optional, but that manual can help you to understand the commands in this recipe, specifically those that start with `\Declare`.

How to do it...

We will copy the required lines of code from the font package to our document. Perform the following steps:

1. From the `mathabx.sty` file, copy all the needed font declarations to your document preamble:

```
\DeclareFontFamily{U}{matha}{\hyphenchar\font45}
\DeclareFontShape{U}{matha}{m}{n}{
  <5> <6> <7> <8> <9> <10> gen * matha
  <10.95> matha10 <12> <14.4> <17.28> <20.74>
  <24.88> matha12
  }{}
\DeclareSymbolFont{matha}{U}{matha}{m}{n}
\DeclareFontSubstitution{U}{matha}{m}{n}
```

2. From the `mathabx.dcl` file, copy all the required font declarations to your document preamble:

```
\DeclareMathSymbol{\leq}{3}{matha}{"A4}
\DeclareMathSymbol{\nleq}{3}{matha}{"A6}
```

3. Test those declarations:

```
If $A \leq B$, then $B \nleq A$.
```

   Take a look at the output:

$$\text{If } A \leqslant B, \text{ then } B \nleqslant A.$$

Compare to the symbols in the default font without our redefinition:

$$\text{If } A \leq B, \text{ then } B \not\leq A.$$

Perhaps you can see a motivation in changing the look of such a specific symbol, regarding parallel lines, writing habits, and vertical centering.

How it works...

By simply copying and pasting, we copied the behavior of the symbol package. We chose only the relevant symbols. The commands are described in detail in the previously mentioned `fntguide` documentation; refer to it for details. Here, we primarily need to copy the exact data from the package source code, because it's designed that way.

We redefined the original `\leq` command, since we don't need to have two versions, as it's recommendable to decide for only one for consistency. However, you can freely choose a name, for example:

```
\DeclareMathSymbol{\myleq}{3}{matha}{"A4}
```

Writing bold mathematical symbols

There are several ways of getting bold mathematical symbols. A classic way is directly provided by LaTeX. Take a look at this code:

```
\boldmath $y=f(x)$\unboldmath
```

It works in the following way:

1. In text mode, we switch to bold math alphabets.
2. We enter the math mode, in which bold symbols are always chosen, if available.
3. We leave the math mode.
4. While in the text mode, we switch the math alphabets back to normal—nonbold.

But that's for making all symbols of a formula bold. Such kind of emphasizing is rather rare today, as it destroys the uniform grayness of the text from a typographer's point of view.

A more common requirement is to get bold versions of certain symbols. For example, bold symbols are often used for vectors and number systems.

In this recipe, we will take the most recommended approach to get bold symbols.

How to do it...

We will use the `bm` package as follows:

1. Load the `bm` package in your preamble. Do this *after* loading all the packages that define symbol fonts because the package works on a higher level. If you are not sure about this, enter the following line of code at the end of all font packages:

   ```
   \usepackage{bm}
   ```

2. Declare a command for each bold symbol:

   ```
   \bmdefine{\balpha}{\alpha}
   \bmdefine{\bX}{X}
   ```

3. Use the new macros in your document within the math mode.

How it works...

The `bm` package does the following for us:

- Determining available bold math fonts and using them if available
- Falling back to *poor man's bold* if no bold version can be found, which means overprinting with slight offsets
- Keeping the correct spacing of the symbol
- Respecting the meaning of symbols, such as delimiters

As it has been made for math symbols, the new bold symbols also can only be used in the math mode.

There's more...

There are alternative ways for writing bold symbols. Let's take a look at some of these.

Standard LaTeX

- One standard command is `\mathbf{argument}`, which prints the argument in bold. However, there are a few drawbacks:
  - It switches from italics to the upright shape. So, what originally had the look of a variable, gets the appearance of an operator
  - It doesn't support many special characters; for example, `\mathbf{\alpha}` prints a normal nonbold Greek alpha letter

AMS-LaTeX and amsmath

- The `amsmath` bundle, specifically, its `amsbsy` package, provides the `\boldsymbol{argument}` command, which also prints its argument in bold. It works for many more symbols, such as for Greek letters. Even if there's no bold version available, you can use `\pmb{argument}`, which provides a *poor man's bold* version of the argument.

Comparing bm and amsmath

This recipe recommends `bm`, which at first sight looks quite similar to `\boldsymbol` because it takes special care of the meaning and spacing. For a visual comparison, let's build a formula with many bold symbols to see the effects.

Take the following example into your LaTeX editor:

```
\documentclass{article}
\usepackage{bm}
\bmdefine{\bX}{X}
\bmdefine{\bi}{i}
\bmdefine{\bMinus}{-}
\bmdefine{\bSum}{\sum}
\bmdefine{\bLeft}{(}
\bmdefine{\bRight}{)}
\begin{document}
\[ \sum_{i} ( - X_{i} ) \]
\[ \bSum_{\bi} \bLeft \bMinus \bX_{\bi} \bRight \]
\end{document}
```

Compile that example, so we can compare the normal and bold versions:

$$\sum_{i}(-X_i)$$

$$\sum_{\boldsymbol{i}}(-\boldsymbol{X_i})$$

Now, we switch to the `amsmath` version:

```
\documentclass{article}
\usepackage{amsmath}
\newcommand{\bX}{\boldsymbol{X}}
\newcommand{\bi}{\boldsymbol{i}}
\newcommand{\bMinus}{\boldsymbol{-}}
\newcommand{\bSum}{\boldsymbol{\sum}}
\newcommand{\bLeft}{\boldsymbol{(}}
\newcommand{\bRight}{\boldsymbol{)}}
\begin{document}
\[ \sum_{i} ( - X_{i} ) \]
\[ \bSum_{\bi} \bLeft \bMinus \bX_{\bi} \bRight \]
\end{document}
```

The output is as follows:

$$\sum_{i}(-X_i)$$

$$\sum\nolimits_{\boldsymbol{i}}(\boldsymbol{-}\boldsymbol{X_i})$$

You can clearly see what happened:

- The summation symbol is not bold.
- The summation index has moved to another position.
- The spacing between the parenthesis and minus sign has become too wide.
- The italic correction after the X is lost as the kerning is too high. That is, the index is too far at the right now.

It's not because of combining bold symbols, it also happens in normal interaction with bold and non-bold symbols. Some of the meanings of symbols and of clever TeX spacing are gone.

If you already work with `amsmath` and `\boldsymbol`, switching to `bm` is very easy: simply add `\usepackage{bm}` after `\usepackage{amsmath}`. The `bm` package redefines `\boldsymbol` to become its own version. This way, all flaws in the previous `amsmath` example will immediately be fixed.

Getting the sans serif mathematics font

There are situations where the sans-serif font is required for documents. It can be, for example, a requirement by the university or institute. It may even be a design decision, for example, presentation slides often use sans serif. It's the default behavior of the LaTeX `beamer` class.

You can switch to sans serif for the default text font family using this command:

```
\renewcommand{\familydefault}{\sfdefault}
```

In such a case, it's desirable to print math formulas in sans serif as well to get a consistent design. The `beamer` class already does this.

In this recipe, we will do that for an arbitrary class. We will change all math formulas to have a sans serif font.

How to do it...

We will use the `sfmath` package. Follow these steps to get sans serif math formulae:

1. Load the `sfmath` package. Do it after loading the font packages or commands that change the `\sfdefault` command:

   ```
   \usepackage{sfmath}
   ```

2. If you use the default font, that is, Computer Modern, additionally load `sansmathaccent`:

   ```
   \usepackage{sansmathaccent}
   ```

3. In your document, write math formulas as usual. For example, this formula from *Chapter 1*, *The Variety of Document Types*:

   ```
   \[
     \int_a^b f(x)\,\mathrm{d}x \approx (b-a)
     \sum_{i=0}^n w_i f(x_i)
   \]
   ```

4. Compile the document. Take a look at the appearance of the output:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i)$$

The math formulas are now written in sans serif. Looking closer, you can see that the sum operator and the integral sign still have serifs though.

How it works...

The `sfmath` package switches to sans serif math for the whole document. It automatically detects the available sans serif font. That's why it is important to load it after any changes made to `\sfdefault`. For example, if you use Helvetica, at first, write `\usepackage{helvet}` and then write `\usepackage{sfmath}`.

There's an issue with math accents. For example, with the default Computer Modern font, math accents such as `\tilde`, `\dot`, and `\hat` may be slightly misplaced, since the Computer Modern sans serif font doesn't provide the positioning information. The `sansmathaccent` package comes to the rescue. In the specific case of `sfmath` together with Computer Modern, it corrects that behavior. That's why we loaded that too.

There's more...

Instead of automatic font selection, you can let `sfmath` explicitly use a certain font using a package option:

- `cm`: This is used for Computer Modern sans serif.
- `lm`: This is used for Latin Modern sans serif (experimental).
- `helvet`: This is used for Helvetica.
- `cmbright`: This is used for CM-Bright.
- `tx`: This is used for `txfonts`, which is a Times font bundle with Helvetiva-like sans serif set.
- `px`: This is used for `pxfonts`, a Palatino font bundle, also with Helvetiva-like sans serif.

Using the following options, you can gain even more control over fonts:

- `T1experimental`: This is used to get T1 encoded maths (experimental, required for Latin Modern).
- `AlphT1experimental`: This is required for using T1 encoding for `\mathrm`, `\mathbf`, `\mathit`, and `\mathsf`.
- `mathrmOrig`, `mathbfOrig`, `mathitOrig`, `mathsfOrig`: These are used for preserving the original behavior of `\mathrm`, `\mathbf`, `\mathit`, or `\mathsf`, respectively. Otherwise, `\mathrm` can also display sans serif font, for example.
- `slantedGreek`: This is used for getting slanted uppercase Greek letters.

So, a call of `sfmath` can look like this:

```
\usepackage[helvet,slantedGreek]{sfmath}
```

An alternative approach

Instead of `sfmath`, you can load the `sansmath` package:

```
\usepackage{sansmath}
```

Now you can switch on sans serif math using this:

```
\sansmath
```

If you would like to end this setting, you could call `\unsansmath` to stop it. There's even a `sansmath` environment, which switches to sans serif math inside. So this package provides a finer approach for the case that you don't need consistent sans serif math.

Sans serif fonts with direct math support

The easiest way of all is switching to a sans serif font that directly supports mathematics. We will discuss two ways to do this. Of course, this depends on whether you like that font at all.

Arev Sans

Bitstream Vera Sans, which is designed as a sans serif screen font, has been extended to include Greek, Cyrillic, and a lot of mathematical symbols. The result is the `arev` package. All you need to do to get sans serif math and text is to load it:

```
\usepackage{arev}
```

The previous equation now becomes:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i)$$

Read the name **Arev** backwards.

Kepler fonts

We already discussed the Kepler font bundle in our first recipe of this chapter. This is a complete font set that also supports sans serif math. You can get sans serif formulae using the following:

```
\usepackage[sfmath,lighttext]{kpfonts}
```

The previous example with Kepler fonts now looks like this:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\sum_{i=0}^{n} w_i f(x_i)$$

Writing double stroke letters as if on a blackboard

Mathematicians need a lot of symbols for variables, constants, vectors, operators, sets, spaces, and a lot of other objects. So they use small and big Latin and Greek letters, calligraphic letters, or write upright, italic, or bold so that they can be distinguished from each other in the same document.

In a lecture, when writing on a blackboard or a whiteboard, it is difficult to write bold letters. So, double stroke letters showed up. Well, in our documents, we still can simply switch to bold, on the other hand double stroke letters for number systems are already very common. Furthermore, a typographer may like this, since it doesn't destroy the grayness of a text, in contrast to bold symbols.

How to do it...

We will use the `dsfont` package. Follow these steps to get double stroke letters:

1. Load the `dsfont` package:

```
\usepackage{dsfont}
```

2. In your document, use the `\mathds` command:

```
\[
  \mathds{N} \subset \mathds{Z} \subset
  \mathds{Q} \subset \mathds{R} \subset \mathds{C}
\]
```

3. The output will now become:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

How it works...

The `dsfont` package provides double stroke capital letters for the whole alphabet; lowercase letters are not supported. That's a design decision. At some stage, the symbols for 1, h, and k have been added.

There's more...

There are alternative font packages: `amssymb`, `bbold`, and `bbm`. They provide similar symbols. I selected `dsfont` because it matches the style I know from my math studies.

Such symbols, and thousands more, can be found in the Comprehensive LaTeX Symbol List. You can open it by typing the `texdoc symbols` command in the Command Prompt, or you can find it online at `http://texdoc.net/pkg/symbols`.

Enabling the searching and copying of ligatures

In certain cases, two or more consecutive characters are joined to a single glyph. This is called a **ligature**. LaTeX commonly does it for ff, fi, fl, ffi, ffl, and more, depending on the font. That's because font makers designed specific glyphs for certain character combinations.

While it looks fine in print and on screen, there is a caveat—if you copy text from the produced PDF file into another document, such as to a text document or a Word file, the ligatures may appear broken.

Another problem is searching for words containing ligatures in PDF files, which can simply fail as the ligature "ff" is not equivalent to the letter combination "ff".

We will now fix that.

How to do it...

We stick to the commonly used pdfLaTeX. There are several possible ways to fix it. The first way is this:

1. Insert the `glyphtounicode.tex` file into your document's preamble:

   ```
   \input{glyphtounicode}
   ```

2. In the next line, activate the required pdfTeX feature:

   ```
   \pdfgentounicode=1
   ```

How it works...

The `glyphtounicode.tex` file is part of pdfTeX and should be installed on your computer. You can find it using `kpsewhich glyphtounicode.tex`. It contains proper translations to alphabetic presentation, for example:

```
\pdfglyphtounicode{ff}{0066 0066}
\pdfglyphtounicode{fi}{0066 0069}
\pdfglyphtounicode{fl}{0066 006C}
\pdfglyphtounicode{ffi}{0066 0066 0069}
\pdfglyphtounicode{ffl}{0066 0066 006C}
```

By setting `\pdfgentounicode=1`, we enabled the translation.

There's more...

The `cmap` package also makes it possible to search and copy characters for making PDF files using the following command:

```
\usepackage{cmap}
```

The `mmap` package is an extension of `cmap`, which also works with math symbols. You can load it instead of `cmap`:

```
\usepackage{mmap}
```

If those methods don't work with a certain font or encoding, and you still need to fix it, you can take a look at the next recipe.

Suppressing ligatures

Ligatures raise the typographical quality and thus we should retain them. However, there may be a reason to disable them, for example, in verbatim text such as the source code.

Furthermore, it's possible that searching or copying of ligatures in a PDF file would fail, which we discussed in the previous recipe.

How to do it...

We will now see how to disable ligatures. We will use the `microtype` package:

1. Load the `microtype` package:
   ```
   \usepackage{microtype}
   ```
2. Disable ligatures completely:
   ```
   \DisableLigatures{encoding = *, family = * }
   ```
3. If you would like to restrict that feature to a certain font, you can specify it instead, such as:
   ```
   \DisableLigatures{encoding = T1, family = tt* }
   ```
4. You may even suppress just the selected ligatures using the following command. Specify the letter that starts the ligature.
   ```
   \DisableLigatures[f]{encoding = *, family = * }
   ```

How it works...

Suppressing ligatures is one of the many features of the `microtype` package. Using the preceding interface, you can switch off some or all the ligatures. Beware, the text doesn't look so nice anymore. Even the kerning of the font is disabled.

Besides the well-known letter ligatures, LaTeX knows more of them, such as the upside-down exclamation and question mark, double quote marks, guillemots, and wide dashes inserted by `--` and `---`. It can be a good idea to not turn off all ligatures. When all ligatures are turned off, there are still commands for writing those ligatures, such as `\textendash` and `\textemdash` for the wide dashes.

Adding a contour

When a text is printed over a background, it is much more readable with a contour around in overprinting the background. This contour may be white. This way, there would be a nice clearance around the text.

Another utilization would be to improve the visibility of the text with very light color for better reading on a white background.

Let's see how to achieve that.

How to do it...

Our example will use yellow color for chapter headings in a book. This is hardly readable, so we will add a contour to improve it:

1. We will use the `scrbook` class, so start your document with the following:

   ```
   \documentclass{scrbook}
   ```

2. Load the `contour` package. Specify the thickness of the contour and a number that stands for text copies internally used for overprinting.

   ```
   \usepackage{contour}
   \contourlength{1.5pt}
   \contournumber{25}
   ```

3. Define a macro for the new font style for our chapter:

   ```
   \newcommand{\chapterfont}[1]{%
   \contour{black}{\textcolor{yellow}{#1}}}
   ```

4. Apply the style that we just selected:

```
\setkomafont{chapter}{\Huge\chapterfont}
```

5. Now, your chapter headings will be in yellow with a black contour. You can verify it by completing the code to a small compilable document. Add the following to:

```
\begin{document}
\chapter{Introduction}
Text follows.
\end{document}
```

6. Compile the document, and take a look at the output:



Text follows.

How it works...

The package creates a contour by placing copies of the text around it. The thickness of this contour can be specified as seen.

The `\setkomafont` command can use a macro at the end, which takes an argument. This will be the heading text in this case, handed over to our newly-defined macro.

A vector font is required, which is recommended in any case. This should be a so called Type 1 font. With pdfLaTeX, TrueType fonts are supported too.

If your engine supports it, you can even let the contour generate a real outline of the text, instead of using copies, which should result in a better quality. For example, with pdfTeX, you can add package options to enable it:

```
\usepackage[pdftex,outline]{contour}
```

See also

With **XeLaTeX** and **LuaLaTeX**, you can use even system fonts such as TrueType fonts and feature rich OpenType fonts. That's actually LaTeX basing on the new engines called **XeTeX** and **LuaTeX**. Those names are more popular.

The TeX User Group has a good page for starting with XeLaTeX:

`http://www.tug.org/xetex/`

LuaLaTeX is a good choice when you need additional programming features, since it adds the Lua programming language. You can read about it at `http://www.luatex.org`.

4
Working with Images

This chapter contains some recipes for including, positioning, and manipulating images within LaTeX. Specifically, we will cover the following topics:

- Including images with optimal quality
- Automating image positioning
- Manipulating images
- Adding a frame to an image
- Cutting an image to get rounded corners
- Shaping an image like a circle
- Drawing over an image
- Aligning images
- Arranging images in a grid
- Stacking images

Introduction

You can include external images into a LaTeX document. Only a few image file formats are supported by LaTeX. We will discuss supported formats and the differences between them.

After discussing quality aspects, we will take a look at the concept of floating figures for automated positioning for well-balanced text heights on pages.

Then, we will look at some recipes that can be used for making images, especially photos, more decorative.

Finally, we will arrange images with alignment, in a grid, or even on top of each other.

While this chapter is about using images, you can read about creating images in *Chapter 9*, *Creating Graphics*. In this chapter, we will focus on required commands without starting each document with `\documentclass` and ending it by `\end{document}`. You can download full code examples from `http://latex-cookbook.net`.

Including images with optimal quality

First, ensure that your images originally have good quality.

Bitmap images, such as JPG/JPEG and PNG, have a fixed number of pixels, so by scaling they may become blurry or pixelated. You may notice this in the images in this book as the production process required bitmap images instead of the original LaTeX output in PDF format.

Vector images, in contrast, are scalable without loss of quality. You can zoom in and out and they keep looking fine. An example of this is the **Scalable Vector Graphics** (**SVG**) format. It's not natively supported by LaTeX, but it can be converted to PDF or to TikZ (PGF). We will talk about TikZ and PGF in *Chapter 9*, *Creating Graphics*. **Portable Document Format** (**PDF**) and **PostScript** (**PS**) are vector formats, although they are allowed to contain bitmap images. Vector formats should be preferred over bitmap formats.

Today, pdfLaTeX is most widely used. It allows the inclusion of PDF images. Furthermore, the bitmap formats, JPG/JPEG, and PNG can be used. Classic LaTeX, which generates **Device independent file format** (**DVI**) files, only supports **Encapsulated PostScript** (**EPS**). This is a PostScript format with some restrictions; for example, it cannot span several pages. It is intended to be embedded in documents. To make embedding easier, an EPS file contains additional dimension information.

Getting ready

When you produce diagrams or drawings using an external program, always try to export them in a vector format. Often, programs can export to PDF or PS format. There are also printer drivers, which can generate PDF or PS files based on the ghostscript.

If your images are originally bitmap images, such as photos or screenshots, there's usually nothing to gain by simply converting to PDF or EPS. Situations where images can actually be vectorized by interpolating with a tool are pretty rare and are for rather simple images. For example, **Inkscape** has a tracing feature for vectorizing bitmaps.

So, for different types of images, we should prepare differently:

- **Drawings and diagrams**: Save them in a scalable format such as PDF or EPS.
- **Photos**: Click them at high resolution. If the camera saves as JPG, you can directly include it. However, if you would like to edit your images, change them to PNG format, which has a lossless compression, in contrast to JPG.
- **Screenshots**: Save them in PNG format and not as JPG. This is also a bitmap, but PNG preserves the original quality. You can make high-resolution screenshots of smaller dialog windows if you switch to a bigger system font and use the highest screen resolution.

Don't resize the images before including them. The PDF viewer or printer driver will do the scaling depending on the device resolution.

How to do it...

The `graphicx` package is the standard tool, which we will now use. Follow these steps:

1. Choose a supported format. If necessary, convert your image to such a format. With classical LaTeX, which produces DVI output, use EPS format. With pdfLaTeX, use PDF, JPG/JPEG, or PNG format.
2. Load the `graphicx` package once in the document preamble:

   ```
   \usepackage{graphicx}
   ```

3. At the place in the document where the image should appear, insert the following command:

   ```
   \includegraphics{filename}
   ```

   If you need to fit to a certain width or height, add it as an argument (as follows) using half of the width of the text:

   ```
   \includegraphics[width=0.5\textwidth]{filename}
   ```

   Note that you can use absolute values with units such as cm, mm, or in:

   ```
   \includegraphics[width=5cm, height=3cm,
     keepaspectratio]{filename}
   ```

   The preceding command is for limiting both dimensions, but the `keepaspectratio` command is used for preventing distortion of the image.

4. For bitmaps, you may enable interpolation as follows:

   ```
   \includegraphics[interpolate]{filename}
   ```

How it works...

The `\includegraphics` command has a mandatory argument, which is the filename of the image. The name can be used without a file name extension.

Furthermore, the command understands a lot of options in `key=value` form. This way, we specified width and height. More options are shown in the next recipe.

The `interpolate` option activates the interpolation for bitmaps, which is supported in PDF. That is, if you zoom in on a raster image, you won't see big pixels. Instead, a smooth transition between adjacent color values would be applied by a capable PDF reader.

Automating image positioning

When there's not enough free space on a page when you include an image, that image will go to the next page. This will leave white space at the end of the page. You could manually move some text to compensate. But imagine having a large document with many images; manually moving images to balance page breaks could cause a headache. Fortunately, LaTeX provides an automatism for us.

How to do it...

This is a very common way of including images as figures:

1. Use a `figure` environment.
2. Center the content, if desired.
3. Include the image.
4. Add a caption.
5. Add a label for cross-referencing.

A typical command sequence is as follows:

```
\begin{figure}[htbp!]
  \centering
  \includegraphics{filename}
  \caption{Some text}
  \label{fig:name}
\end{figure}
```

In the document, you can refer to the figure number by using the `\label{fig:name}` command.

How it works...

The automatism is called **floating**, and it works for images; that means figures and tables. A `figure` environment lets its content float to the next available place. For example, when a page runs out of space, a figure can move to the top of the next page. Text, which comes after the figure in the document code, would be moved so that it appeared before the figure, to fill the page.

You noticed the preceding options `htbp!`. These characters allow placement *here* (if there is space), at the *top*, at the *bottom*, or on a dedicated *page*. The exclamation mark relaxes some typographical restrictions to ease placement. All options together cause the output of figures as near as possible. Leave out an option to disallow certain places. For example, without option `b`, a figure would not float to the bottom, but may float to the top.

The `\caption` command sets text below the figure and gives it a number. If you use the `\label` command, it has to go after the `\caption` command to get the correct number. The prefix `fig:` has been used because it's good practice for indicating types of cross-reference, such as `tab:` for tables and `eq:` for equations.

There's more

If the automated positioning of images would not be the optimal choice for your document, there are ways to limit it or to temporarily deactivate it. We will take a look at these options now.

Limiting the floating figures and tables

Floating figures don't cross chapter borders. However, they may go to a later section. If you would like to restrict the floating so figures and tables stay within the same section, load the `placeins` package in your preamble with the `section` option:

```
\usepackage[section]{placeins}
```

You can also load the `placeins` package without options. In any case, it gives you a new command. You can now use the command `\FloatBarrier` to block floating beyond this point.



The command `\clearpage` ends the page and forces the output of all floats that are not yet placed.

Fixing the position of a figure

Sometimes figures should stay at a certain position. Disabling floating is easy: simply don't use a `figure` environment. However, if you would like to use the very same syntax as the preceding one, just without floating, you can achieve it by using the following command in the document preamble:

```
\usepackage{float}
```

Now, start a `figure` environment with the `H` option:

```
\begin{figure}[H]
```

See also

There's an extensive document about using imported graphics in LaTeX. It includes a thorough explanation of the concept of floats. It's available on CTAN at `http://mirrors.ctan.org/info/epslatex/english/epslatex.pdf`.

Manipulating images

Before including it in a document, an image should be prepared using graphics software. LaTeX is not the right tool for post-processing of images. However, there are some basic ways to customize the way an image is included.

How to do it...

The `graphicx` package allows customization via simple options:

- By scaling an image, you can specify a scaling factor, like so:

  ```
  \includegraphics[scale=0.5]{filename}
  ```

- You can resize to a fixed width, using width and height options as in the previous recipe.
- You can rotate by specifying a clockwise rotation angle, like so:

  ```
  \includegraphics[angle=90]{filename}
  ```

- You can rotate around a certain origin by adding a key, such as `c` for center, `B` for the base line, and `l`, `r`, `t`, and `b` for left, right, top, and bottom, respectively. A combination would be understood, such as `tl` for top-left corner:

  ```
  \includegraphics[angle=90,origin=tl]{filename}
  ```

- You can trim and clip, like here, where we cut off 1 cm at the left, 2 cm at the bottom, 3 cm at the right, and 4 cm at the top:

  ```
  \includegraphics[trim=1cm 2cm 3cm 4cm,clip]{filename}
  ```

You can read all details in the package manual, which you can access by typing the `texdoc graphicx` command in Command Prompt, or by visiting `http://texdoc.net/pkg/graphicx`.

Adding a frame to an image

You can add a simple frame to an image or to text by using the `\frame{...}`, `\framebox{...}`, or `\fbox{...}` commands. However, this would give a simple box with thin black lines and a certain distance to the content. How about changing the color, line thickness, or distance? The classic way to do the latter is by changing the LaTeX lengths `\fboxrule` and `\fboxsep`. It's a bit laborious, especially when the lengths can vary. It can be made easier.

How to do it...

We will load the `adjustbox` package. It provides several useful commands for modifying boxes. It implicitly loads the `graphicx` package and exports its own features to the `\includegraphics` command. Follow these steps:

1. Load the `xcolor` package:

   ```
   \usepackage{xcolor}
   ```

2. Load the `adjustbox` package with the `export` option:

   ```
   \usepackage[export]{adjustbox}
   ```

3. At the place in your document where the image is to be placed, use the `\includegraphics` command, like in the previous recipe. This time, add the `cframe` option:

   ```
   \includegraphics[width=10cm,
     cframe=red!50!black 5mm]{filename}
   ```

4. Compile the document, like I did with a photo of my dog. Now, you can see a frame fitting right around the image:



How it works...

The `xcolor` package provides commands for specifying colors by name and for mixing color. Here, with its syntax `red!50!black`, we chose 50 percent red color mixed with black.

The `adjustbox` package is capable of exporting some of its features. We activated that export; now we can use additional options with the known `\includegraphics` command. We chose the `cframe` option for a colored frame, which has this syntax:

```
cframe=color thickness separation margin
```

The `color` parameter is mandatory, and the `xcolor` package is required. The other values are optional lengths, but are understood in this order. So, we chose a frame thickness of 5 mm and kept the default zero inner separation and outer margin.

There's a similar `frame` option which works without the color value; that is, without value or up to three lengths given for thickness, separation, and margin.

The companion `cfbox` and `fbox` options work in a similar way, just internally based on the `\fbox` command instead of the `\frame` command. In other words, they use the default thickness of the `\fboxrule` command, and the content separation of the `\fboxsep` command.

Cutting an image to get rounded corners

The previous recipe gave us an edged image. It can be nice to have rounded corners, so let us take a look at how to achieve that.

How to do it...

We will use a few features of the very capable `tikz` graphics package:

1. Load the `tikz` package in your preamble:

   ```
   \usepackage{tikz}
   ```

2. Declare a box for storing the image:

   ```
   \newsavebox{\picbox}
   ```

3. Define a macro that allows us to use our recipe repeatedly:

   ```
   \newcommand{\cutpic}[3]{
     \savebox{\picbox}{\includegraphics[width=#2]{#3}}
     \tikz\node [draw, rounded corners=#1, line width=4pt,
       color=white, minimum width=\wd\picbox,
   ```

```
    minimum height=\ht\picbox, path picture={
      \node at (path picture bounding box.center) {
        \usebox{\picbox}};
    }] {};}
```

4. Use the new macro within your document to include an image:

   ```
   \cutpic{1cm}{8cm}{filename}
   ```

5. Compile the document to see the effect:



How it works...

After loading the `tikz` package, we created a box to store the image. We defined a macro; its first task is to store our image in the box by using the `\savebox` command.

The new macro `\cutpic` takes three arguments:

- A length for the rounding value
- The width of the image
- Its filename

The command `\tikz` is an abbreviation of the `\begin{tikzpicture}` ... `\end{tikzpicture}` command, which is handy for simple figures. Here, the figure is just a single node with some options.

The option `rounded corners=width` produces the shape of a rounded rectangle for the node. We chose to let it draw in white color with 4 pt thickness. At this place, you can choose a color you like.

While the node has an empty node text, we required a minimum height and width to match the size of our image. Now our box comes into play: we measured its width and height for handing over to the node with the TeX primitive commands `\wd` and `\ht`.

Finally, we used an advanced option of TikZ. The `path picture=code` command fills a path of the drawing by executing the given code. The result of the code will be clipped by that path. So, instead of filling with a color, a pattern, or a shade, we printed out the image by using the `\usebox` command. For this, we encapsulated the image in its own node, and placed it at the center.

There's more...

TikZ usually offers more than one way of doing something. An alternative approach would be to use the `current bounding box` node to define a clip path, or to simply draw a rectangle with rounded corners in white color over the image.

Using nodes has the advantages that you will get anchors for alignment, for connecting by edges or arrows that you can name it and refer to it, and that you get shapes and further options right away. You can also use it for alignment, which we will demonstrate in the next recipe.

In *Chapter 9*, *Creating Graphic*s, you will learn more about TikZ.

Shaping an image like a circle

A circle shape is a neat option for portrait photos or for images arranged in a graph or a tree.

How to do it...

Like in the previous recipe, we will define a TikZ macro for this purpose. Let's take a look at the following steps:

1. Load the `tikz` package:

```
\usepackage{tikz}
```

2. Define a macro so we can use it often:

```
\newcommand{\roundpic}[4][]{
  \tikz\node [circle, minimum width = #2,
    path picture = {
      \node [#1] at (path picture bounding box.center) {
```

```
        \includegraphics[width=#3]{#4}};
    }] {};}
```

3. Use the new macro within your document to include an image:

```
\roundpic[xshift=-1cm,yshift=-2.6cm]{5.8cm}{9cm}{filename}
```

4. Compile the document. When I used the photo of my dog, I got the following image:



How it works...

Refer to the previous recipe to understand the TikZ node construction with the `path picture` command. This time, we used a circle for the outer node, so our image got cropped in to a circular shape. We used four arguments. The first optional argument can contain node options. In our example, we used them to shift the image for better positioning in the cut window. The remaining options are the width of the node, the width of the image, and, of course, the name of the image file.

There's more...

When we normally include images, the base line is at the bottom. We can change it to be at the center, for positioning. In such a case, we can give the node a name and then tell TikZ to use the center of the node as the baseline for aligning the whole image, as follows:

```
\newcommand{\roundpic}[4][]{
  \tikz[baseline=(photo.center)]
    \node (photo) [circle, minimum width = #2,
      path picture = {
        \node [#1] at (path picture bounding box.center)
        {\includegraphics[width=#3]{#4}};
}] {};}
```

Drawing over an image

If you need to add text, arrows, or other annotations to an image, it's preferable to do so within LaTeX. Compared to using external graphics software, there are a few advantages:

- You can use the same fonts as in the LaTeX document
- Your annotations would be scalable in perfect quality, even if you would draw over a bitmap image
- You can use macros from your preamble or packages
- The style would be consistent with your other drawings and diagrams
- You can change it at any time, and document-wide adjustments also have their desired effects on this annotated image

How to do it...

We will draw with TikZ. The key is using the `onimage` package. If this is not available in your TeX distribution or on CTAN, you can download it from Launchpad at `http://bazaar.launchpad.net/~tex-sx/tex-sx/development/view/head:/onimage.dtx`.

Follow these steps:

1. Within the LaTeX document preamble, load the `onimage` package:

```
\usepackage{onimage}
```

2. Define TikZ styles for your annotations:

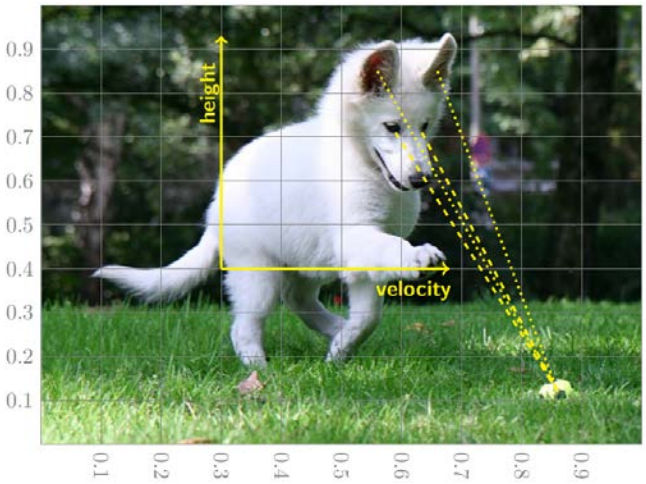
```
\tikzset{annotations/.style = {
  tsx/show help lines,
  every path/.append style = {very thick, color = yellow},
  every node/.append style = {yellow,
                              font = \bfseries\sffamily}}}
```

3. In your document body, use the `tikzonimage` environment:

```
\begin{tikzonimage}[width=.8\textwidth]{filename}
  [annotations]
  \draw[dashed] (0.59,0.71)  -- (0.86,0.12)
                (0.634,0.71) -- (0.86,0.12);
  \draw[dotted] (0.56,0.85)  -- (0.86,0.12)
                (0.66,0.85)  -- (0.86,0.12);
  \draw (0.3,0.4) edge[->] (0.68,0.4)
        (0.3,0.4) edge[->] (0.3,0.93);
```

```
  \node[rotate=90] at (0.28,0.8) {height};
  \node            at (0.62,0.35) {velocity};
\end{tikzonimage}
```

4. Compile, like I did with a photo of my dog. Now you can see the additional illustrations:



How it works...

You can download a complete code example at `http://latex-cookbook.net`. Loading the `onimage` package automatically loaded the `tikz` graphics package. Using the `\tikzset` command, we defined a style named `annotations`, so we can easily apply it at several places. In this style, we activated the `tsx/show help lines` option, which draws a grid over the image. We can use this later for getting coordinates visually. For the final document, we comment out or delete this option to remove the grid.

Furthermore, we defined that every path in such annotations should be drawn in yellow and very thick, and all text in nodes should have a bold sans serif font in yellow color. Since we did this only for the `annotations` style, other drawings won't be affected.

For the image itself, we used the `tikzonimage` environment. Its syntax is as follows:

```
\begin{tikzonimage}[image options]{filename}[TikZ options]
  ... your TikZ code ...
\end{tikzonimage}
```

This handy environment has the following features:

- It implicitly creates a `tikzpicture` environment with the given TikZ options
- It includes the image within a node, passing the `image options` and `filename` parameters to the `\includegraphics` command
- It gives a coordinate system with the origin at the lower-left corner of the image
- It makes the coordinate system relative to the image size; that is, the point (1,1) or the upper-right corner of the image
- It draws a grid of help lines, if desired

As this has already been done for us, we can concentrate on the actual drawing commands. We used standard TikZ syntax:

- Drawing dashed and dotted lines between coordinates in parentheses; all lines inherit the `very thick` and `yellow` styles
- Drawing arrows, which are edges with an arrow shape, done by the `->` option
- Placing a rotated node and a normal node with text, inheriting the yellow bold sans serif font from the `annotations` style

The grid helps to get the desired coordinates for perfect placement, adjust the estimated coordinates, and compile again. Since the coordinates are relative, scaling the whole image doesn't matter.

You may additionally use further TikZ features, including shapes, arrows, colors, fadings, and transparency. Some of them you will see in *Chapter 9*, *Creating Graphics*.

Aligning images

By default, the baseline of an image is at its bottom. So, adjacent images would be aligned at the bottom. However, alignment at the top, or vertically centered alignment, may be desirable instead.

How to do it...

We will use the `\height` macro for shifting to get vertical centering. Let's take a look at the following steps:

1. Load the `graphicx` package. For testing, or if you don't have images, add the `demo` option to use black rectangles in place of images:

   ```
   \usepackage[demo]{graphicx}
   ```

2. In your document, use the `\raisebox` command, together with half of the `\height` command:

```
\raisebox{-0.5\height}{\includegraphics[height=4cm,
  width=8cm]{filename1}}
\hfill
\raisebox{-0.5\height}{\includegraphics[height=2cm,
  width=4cm]{filename2}}
```

3. Your images, in the demo case, black-filled rectangles, will be vertically centered aligned:



How it works...

The `\height` command returns the current height above the base line. The `\totalheight` command would include the height below the base line, which doesn't exist in our case.

We raised each box by minus half of its height, so we got vertical centering. Even text before or after would be aligned to the middle, because we lowered all the boxes.

Writing the following command would give us top alignment of the images:

```
\raisebox{-\height}{\includegraphics{...}}
```



The `adjustbox` package provides a lot of user commands for aligning boxes with text or images.

Arranging images in a grid

A larger set of photos, plots, or diagrams could be loaded by using the `\includegraphics` command with some space in between, possibly arranged using the `minipage` environments. A `for` loop may help if the file names can be generated.

In this recipe, we will produce a grid of aligned images with arbitrary names easy to arrange.

How to do it...

We will use a `tabular` environment for positioning. That's no surprise yet. However, we will read in the `tabular` cell content, which we will then use as file names for inclusion. The `collcell` package provides the required feature. Let's take a look at the following steps:

1. Load the `graphicx` package and the `collcell` package:

```
\usepackage{graphicx}
\usepackage{collcell}
```

2. Define a command for including an image with chosen width and height:

```
\newcommand{\includepic}[1]{\includegraphics[width=3cm,
  height=2cm,keepaspectratio]{#1}}
```

3. Define a new column type that uses that new command in its column specification. We use the letter `i` for image:

```
\newcolumntype{i}{@{\hspace{1ex}}
  >{\collectcell\includepic}c<{\endcollectcell}}
```

4. In your document, use a `tabular` environment with the new column type. Again, photos of a small dog will serve as an example, because he doesn't complain. The cells contain the basic file name; that is, `meadow` for `meadow.jpg`, and so on:

```
\begin{tabular}{iii}
  meadow  & sea     & beach \\
  blanket & tunnel  & tired \\
  pond    & chewing & halfasleep
\end{tabular}
```

5. Compile the document and check out the result:



How it works...

For automation, we used the `\newcolumntype` command from the `array` package. This package is implicitly loaded by using the `collcell` package. `@{\hspace{1ex}}` is a command for changing the intercolumn space to `1ex`. This is useful for adjusting the `tabular` spacing. The `array` syntax `>{...}` inserts code before a cell, while the `<{...}` syntax inserts code after it.

However, we cannot simply write the `>{\includegraphics{}}` and `<{}}` syntax just to enclose the filename in the `\includegraphics{...}` command. We cannot avoid the fact that the compiler will take the curly braces literally, mixing array syntax with the `\includegraphics` command argument braces.

The `collcell` package comes to the rescue. Its command `\collectcell` starts reading in the cell content. Its command `\endcollectcell` states the end of that content, which will be provided to the user's macro. The latter is the argument for the `\collectcell` command; in our case, this is the `\includepic` command. We defined this macro to encapsulate the `\includegraphics` command together with our desired options, including the image width.

Now we can simply enter image file names in columns, given that we used this column type. Of course, it can be combined with standard column types.

Stacking images

We can also stack images on top of each other, such as for a fancy photo collage. This can be combined with the previous recipes to do things like rotating and framing. Let's take a look at pure stacking.

How to do it...

The `stackengine` package allows the placement of things above each other. It can handle text and math as well as images. Let's try it with the latter, using sample images. Follow these steps:

1. In your document preamble, load the `mwe` package. It provides dummy images and automatically loads the `graphicx` package, which we would otherwise load ourselves, as before:

   ```
   \usepackage{mwe}
   ```

2. Load the `stackengine` package:

   ```
   \usepackage{stackengine}
   ```

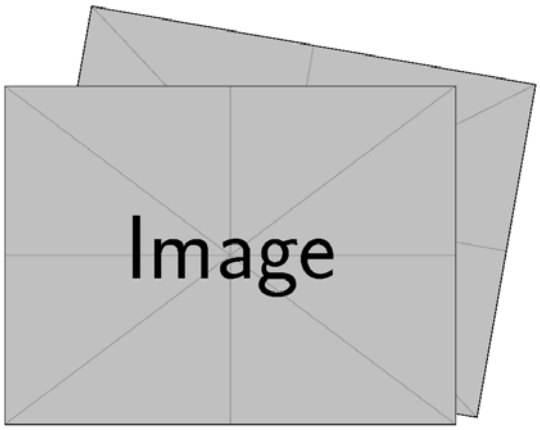
3. In the document body, use the command `\stackinset`. It takes six arguments. This sounds complicated, but it allows flexible positioning. The syntax is as follows:

   ```
   \stackinset{horizontal alignment}
       {horizontal offset}
       {vertical aligment}
       {vertical offset}
       {image above}{image below}
   ```

   In our recipe, we use right and top alignment and shift by `2cm` horizontally and vertically:

   ```
   \stackinset{r}{2cm}{t}{2cm}{%
     \includegraphics{example-image}}{%
     \includegraphics[angle=-10]{example-image}}
   ```

4. Compile the document, and you will get the following result:



How it works...

Let's break down the preceding syntax:

- Horizontal alignment can be `l` for left, `c` for center, or `r` for right aligned. In the case of right alignment, the offset means shifting to the left by this length, or otherwise to the right.
- Vertical alignment can be `t` for top, `c` for center, or `b` for bottom aligned. In the case of top alignment, the offset means shifting down by this length, or otherwise upwards.

You can also specify negative lengths or leave it empty for zero offset.

Besides this, for images, you can use the `\stackinset` command for stacking letters and symbols, for example `\stackinset{c}{}{c}{}{$\star$}{O}`:

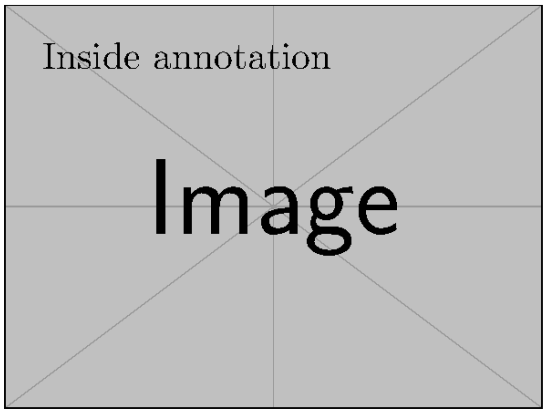
This produces a big letter O with a star inside:



You can mix images and text. This way you could place any annotation over an image, by using commands such as the following:

```
\stackinset{l}{1em}{t}{1em}{Inside annotation}{%
  \includegraphics[width=5cm]{example-image}}
```

This results in the following image:



You can nest several `\stackinset` commands for placing several annotations or for stacking several images.

To see all the features in detail, take a look at the `stackengine` manual. You can access it by typing the `texdoc stackengine` command in Command Prompt, or by visiting `http://texdoc.net/pkg/stackengine`.

5
Beautiful Designs

This chapter contains the following recipes with design ideas:

- Adding a background image
- Creating beautiful ornaments
- Preparing pretty headings
- Producing a calendar
- Mimicking keys and menu items

Introduction

Non-standard documents, such as photo books, calendars, greetings cards, and fairy tale books, may have a fancier design. The recipes in this chapter will offer some decorative examples.

Adding a background image

We can add background graphics, such as watermarks, pre-designed letter-heads, or photos to any LaTeX document. This recipe will demonstrate how.

How to do it...

We will use the `background` package. In this recipe, you can use any LaTeX document. You may also start with the `article` class and add some dummy text. The code bundle for the book contains full examples. You can download it at `http://latex-cookbook.net`.

You just need to insert some commands into your document preamble, which means between the `\documentclass{…}` and `\begin{document}` commands. It would do the following:

- Load the background package
- Set up the background using the command `\backgroundsetup`, with options

Here we go:

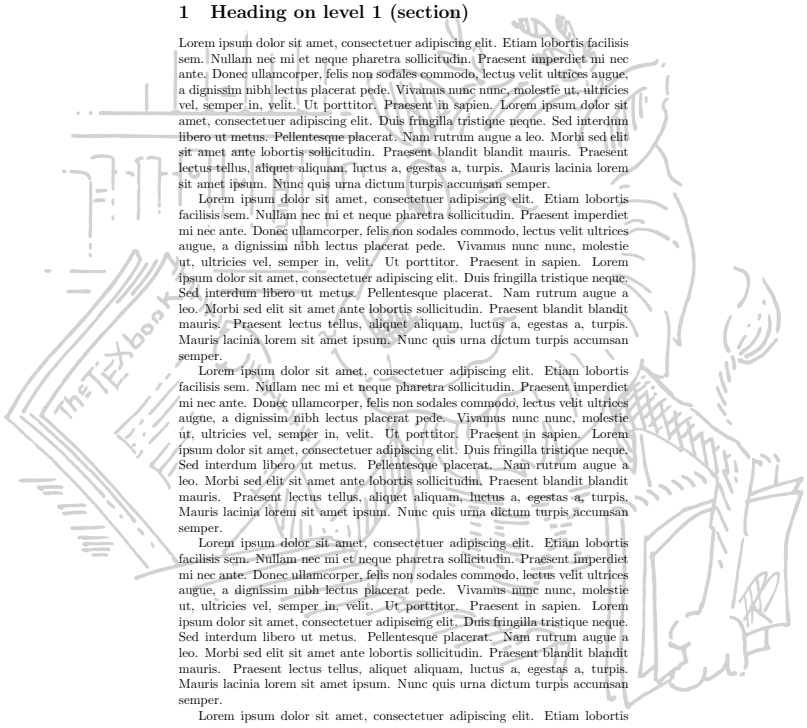
1. Load the `background` package:

   ```
   \usepackage{background}
   ```

2. Set up the background. Optionally, specify scaling factor, rotation angle, and opacity. Provide the command for printing in the background. We will use the `\includegraphics` command here, with a drawing of the CTAN lion:

   ```
   \backgroundsetup{scale = 1, angle = 0, opacity = 0.2,
     contents = {\includegraphics[width = \paperwidth,
     height = \paperheight, keepaspectratio] {ctanlion.pdf}}}
   ```

3. Compile the document at least twice to let the layout settle. Now, all of your pages will show a light version of the image over the whole page background, like this:



1 Heading on level 1 (section)

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis

1

How it works...

The `background` package can place any text, drawing, or image, on the page background. It provides options for position, color, and opacity. The example already showed some self-explanatory parameters. They can be given as package options or by using the `\backgroundsetup` command. This command can be used as often as you like to make changes.

The `contents` option contains the actual commands, which will be applied to the background. This can simply be `\includegraphics`, some text, or any sequence of drawing commands.

The package is based on TikZ and the `everypage` package. It can require several compiling runs before the positioning is finally correct. This is because TikZ writes the position marks into the `.aux` file, which gets read in and processed in the next LaTeX run.

There's more...

Instead of images, you could display dynamic values such as the page number or the head mark with chapter title, instead of using a package such as `fancyhdr`, `scrpage2`, or `scrlayer-scrpage`.

The following command places a page number on the background:

```
\backgroundsetup{placement = top, angle = 0,
  scale = 4, color = blue!80,vshift = -2ex,
  contents = {--\thepage--}}
```

The preceding command does the following with the page number:

- It is placed at the top
- It is placed with customizable rotation, here 0 degrees
- It is scaled four times the size of normal text
- It is colored with 80 percent of standard blue (like mixed with 20 percent of white)
- It is vertically shifted by `2ex` downwards
- It is placed with dashes around

Here is a cut-out of the top of page 7:

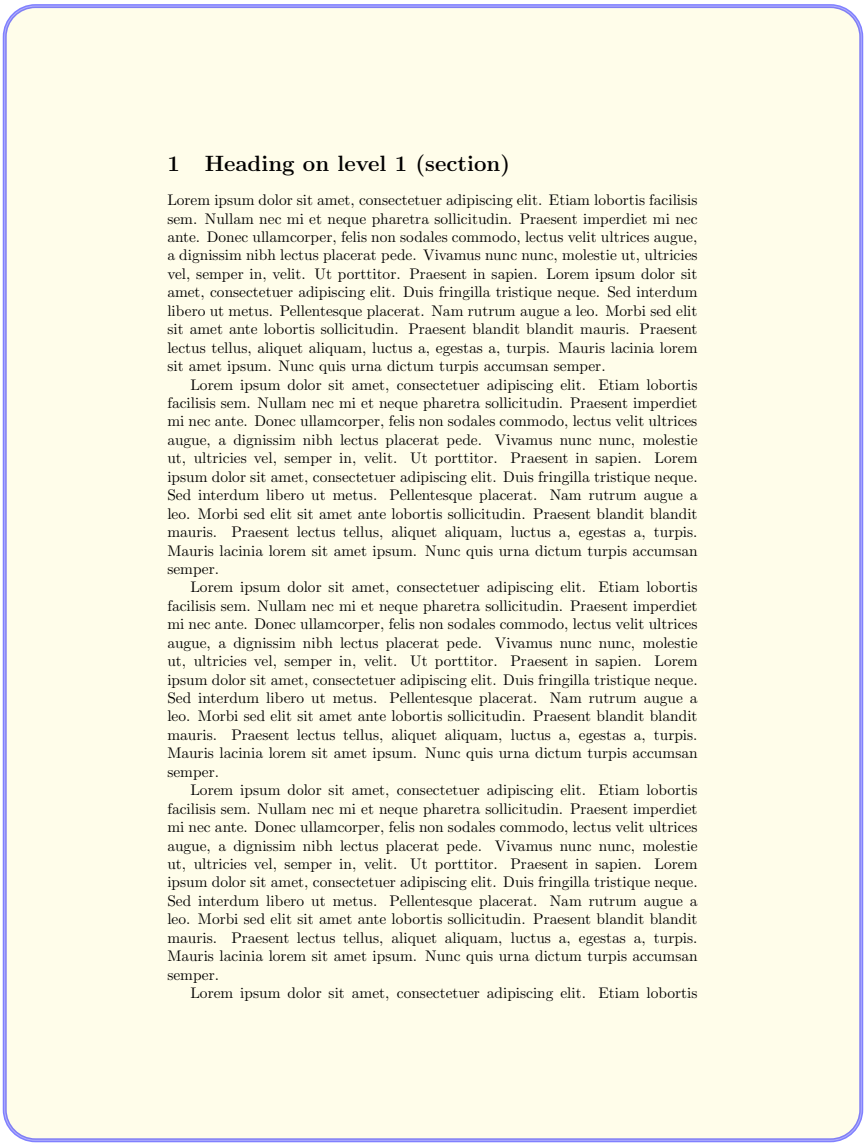


placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

To demonstrate how you can draw with TikZ on the background, let's take a look at an example. The following commands draw a rounded border and fill the interior background with light yellow color:

```
\usetikzlibrary{calc}
\backgroundsetup{angle = 0, scale = 1, vshift = -2ex,
  contents = {\tikz[overlay, remember picture]
    \draw [rounded corners = 20pt, line width = 1pt,
           color = blue, fill = yellow!20, double = blue!10]
           ($(current page.north west)+(1cm,-1cm)$)
           rectangle ($(current page.south east)+(-1,1)$);}}
```

Here, we first loaded the `calc` library, which provides syntax for the coordinate calculations that we used at the end. A TikZ image in `overlay` mode draws a rectangle with rounded corners. It has double lines with yellow in between. The rectangle dimensions are calculated from the position of the `current page` node, which stands for the whole page. The full code example can be downloaded at `http://latex-cookbook.net`. The result looks like this:



1 Heading on level 1 (section)

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis

Here is a summary of selected options with their default values:

- `contents`: This covers text, images, or drawing commands. `Draft` is the default.
- `placement`: This has the `center`, `top` or `bottom` options. `center` is the default.

- `color`: A color expression that TikZ understands. `red!45` is the default.
- `angle`: This holds a value between -360 and 360. `0` is the default for `top` and `bottom`; `60` for `center`.
- `opacity`: This is a value for the transparency between `0` and `1`. The default is `0.5`.
- `scale`: This is a positive value. The default is `8` for `top` and `bottom` and `15` for `center`.
- `hshift` and `vshift`: Any length for horizontal or vertical shifting. The default is `0 pt`.

Further options for TikZ node parameters are explained in the package manual, which also contains some examples. It also shows how to select only certain pages to have this background. You can open it by typing the `texdoc background` command into the command line or by visiting `http://texdoc.net/pkg/background`.

There are more packages that can perform tasks similar to those covered in this recipe; for example, `watermark`, `xwatermark`, and the packages `everypage` and `eso-pic`, which don't require TikZ.

Creating beautiful ornaments

Especially in older books, we can find typographic ornaments such as calligraphic flowers. In this recipe, we will create a greetings card with ornaments.

Getting ready

We will use the `pgfornaments` package. At the time of writing, it was not yet part of a TeX distribution. This may change soon; until then, you can visit the package homepage (`http://altermundus.com/pages/tkz/ornament/index.html`) to download and for install information. Install it before you move on.

How to do it...

We will use a KOMA-Script class, because of its arbitrary base font size. The `calligra` package provides a font with a hand-written appearance:

1. Load the `document` class, set paper and text dimensions, and choose an empty page style, so there's no page number printed automatically:

```
\documentclass[paper=a6,landscape,fontsize=30pt]{scrartcl}
\areaset{0.9\paperwidth}{0.68\paperheight}
\pagestyle{empty}
```

2. Activate `T1` font encoding and load the `calligra` font package:

```
\usepackage[T1]{fontenc}
\usepackage{calligra}
```

3. Load the `pgfornaments` package. It implicitly uses TikZ. In addition, load the `calc` library, which will be used for coordinate calculations:

```
\usepackage{pgfornament}
\usetikzlibrary{calc}
```

4. Now, write the document body:

```
\begin{document}
\centering
\begin{tikzpicture}[
    every node/.style       = {inner sep = 0pt},
    pgfornamentstyle/.style = {color     = green!50!black,
                               fill      = green!80!black}]
  \node [text width = 8cm, outer sep = 1.2cm,
    text centered, color = red!90!black] (Greeting)
    { \calligra Happy Birthday,\\Dear Mom!\\[-1ex]
      \pgfornament[color = red!90!black,
        width = 2.5cm]{72}};
  \foreach \corner/\sym in {north west/none, north east/v,
      south west/h, south east/c} {
      \node [anchor = \corner] (\corner)
        at (Greeting.\corner)
        {\pgfornament[width = 2cm, symmetry = \sym]{63}};}
  \path (north west) -- (south west)
          node [midway, anchor = east]
               {\pgfornament[height = 2cm]{9}}
        (north east) -- (south east)
          node [midway, anchor = west]
               {\pgfornament[height = 2cm,
                symmetry = v]{9}};
  \pgfornamenthline{north west}{north east}{north}{87}
  \pgfornamenthline{south west}{south east}{south}{87}
\end{tikzpicture}
\end{document}
```

5. Compile the document, and take a look at the output:



How it works...

The `pgfornaments` package currently provides 89 vector ornaments. They can be scaled while maintaining high quality. There are calligraphic flowers, tree leafs, and generic symbols and lines that can be used to achieve this kind of vintage design. The documentation contains a lot of information about how it's been done and how it can be used.

The basic command is `\pgfornament[options]{number}`. `number` stands for the chosen ornament, numbered according to the order in which they are listed in the package manual. The options can be a comma separated list of expressions. Let's take a look at the following options:

- `scale`: This can be a positive value; by default, it is `1`
- `width` and `height`: These are LaTeX lengths
- `color`: This can be any color understandable to TikZ
- `ydelta`: This can be set to vertically shift
- `symmetry`: This can get a value `v`, `h`, `c`, or `none` to get the ornament with vertical, horizontal, central, or no mirroring

In our example, we used it in this way:

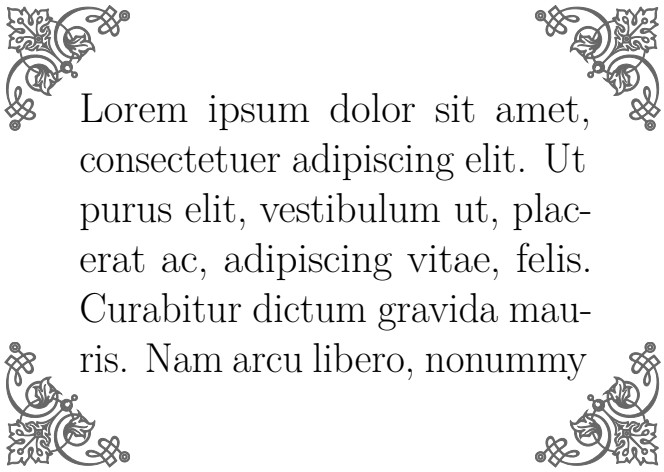
- We set a TikZ style which as an `inner sep` value of 0 pt, so that nodes don't have additional whitespace around them.
- We set the `pgfornamentstyle` parameter to be dark green and filled with lighter green.
- A node that we called `Greeting` contains the handwritten text in dark red Calligra font. In addition, it contains the ornament number 72.
- Using the TikZ `\foreach` syntax, we placed the ornament number 63 at each corner with a suitable symmetry, with relative positioning to our `Greeting` node.
- We placed the ornament number 9 on the left- and right-hand sides. We used the `midway` option to place them right in the middle of the stated corners.
- Finally, the command `\pgfornamenthline` is used to place ornament line number 87 on the top and bottom edges.

There's more...

In the previous recipe, we talked about the `background` package for printing things on the page background. You can combine this with ornaments. This code in your preamble will print neat triangular ornaments with suitable symmetry in each corner of the page:

```
\usepackage{pgfornament}
\usepackage{background}
\backgroundsetup{angle = 0, scale = 1, opacity = 1,
  color = black!60,
  Contents = {\begin{tikzpicture}[remember picture, overlay]
    \foreach \pos/\sym in {north west/none, north east/v,
      south west/h, south east/c} {
        \node[anchor = \pos] at (current page.\pos)
          {\pgfornament[width = 2cm, symmetry = \sym]{63}};}
  \end{tikzpicture}}}
```

Let's take a look at the output:



We used the `current page` node for positioning relative to the page corners. Also, this one requires several compiler runs to achieve final placement. A full code sample is in the code bundle for this book. You can download it at `http://latex-cookbook.net`.

There are fonts which provide typographical ornaments as glyphs; `fourier-orns`, `adforn`, and `webomints` are very good examples.

Preparing pretty headings

This recipe will showcase how to bring some color into document headings.

How to do it...

We will use TikZ for coloring and positioning. Follow these steps:

1. Set up a basic document with blindtext support:

```
\documentclass{scrartcl}
\usepackage[automark]{scrpage2}
\usepackage[english]{babel}
\usepackage{blindtext}
```

2. Load the `tikz` package beforehand, and pass a naming option to the implicitly loaded package `xcolor` to use names for predefined colors:

```
\PassOptionsToPackage{svgnames}{xcolor}
\usepackage{tikz}
```

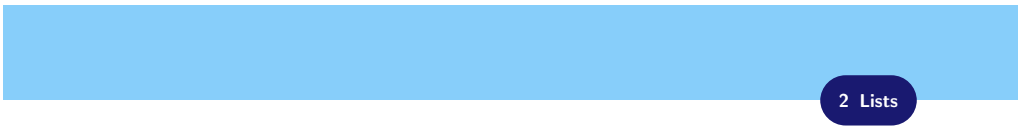
3. Define a macro that prints the heading, given as an argument:

```
\newcommand{\tikzhead}[1]{%
  \begin{tikzpicture}[remember picture,overlay]
    \node[yshift=-2cm] at (current page.north west)
      {\begin{tikzpicture}[remember picture, overlay]
          \path[draw=none, fill=LightSkyBlue] (0,0)
                rectangle (\paperwidth,2cm);
          \node[anchor=east,xshift=.9\paperwidth,
            rectangle, rounded corners=15pt,
            inner sep=11pt, fill=MidnightBlue,
                    font=\sffamily\bfseries] {\color{white}#1};
    \end{tikzpicture}
  };
\end{tikzpicture}}
```

4. Use the new macro for the headings, printing `\headmark`, and complete the document with some dummy text:

```
\clearscrheadings
\ihead{\tikzhead{\headmark}}
\pagestyle{scrheadings}
\begin{document}
\tableofcontents
\clearpage
\blinddocument
\end{document}
```

5. Compile the document and take a look at a sample page header:



ii. Second item in a list

b) Second item in a list

2. Second item in a list

2.3 Example for list (description)

First item in a list

Second item in a list

How it works...

We created a macro that draws a filled rectangle over the whole page width and puts a node with text inside it, shaped as a rectangle with rounded corners. It's just a brief glimpse at the TikZ drawing syntax, which we will explore in more detail in *Chapter 9*, *Creating Graphics*.

The main points are as follows:

- Referring to the current page node for positioning, as in the first recipe of this chapter
- Using the drawing macro within a header command

The rest are drawing syntax and style options, described in the TikZ manual. You can read it by typing the `texdoc tikz` command in Command Prompt or by visiting `http://texdoc.net/pkg/tikz`.

Producing a calendar

Self-made calendars can be a great gift. Also, for work and education, it would be great to have our own customized calendar.

In this recipe, we will print a calendar of a whole year, with all months arranged in a tabular layout. You can adjust it to print just one month below an image, for example.

How to do it...

We will use the mighty TikZ bundle again, since it provides a calendar library:

1. Set up `document` class and page dimensions. Furthermore, change to `empty` pagestyle to not have page numbering:

```
\documentclass{article}
\usepackage[margin = 2.5cm, a4paper]{geometry}
\pagestyle{empty}
```

2. Load TikZ and its libraries `calendar` and `positioning`:

```
\usepackage{tikz}
\usetikzlibrary{calendar,positioning}
```

3. To save the typing effort and for an easier change of year, define a macro for the year and one to call the TikZ `\calendar` command:

```
\newcommand{\calyear}{2016}
\newcommand{\mon}[1]{\calendar[dates = \calyear-#1-01
  to \calyear-#1-last] if (Sunday) [red];}
```

4. Now write the document containing a TikZ picture. The months' calendars are arranged in a matrix:

```
\begin{document}
  \begin{tikzpicture}[every calendar/.style = {
      month label above centered,
      month text = {\Large\textsc{\%mt}},
      week list,
    }]
    \matrix (Calendar) [column sep = 4em, row sep = 3em] {
      \mon{01} & \mon{02} & \mon{03} \\
      \mon{04} & \mon{05} & \mon{06} \\
      \mon{07} & \mon{08} & \mon{09} \\
      \mon{10} & \mon{11} & \mon{12} \\ };
      \node [above = 1cm of Calendar, font = \Huge]
        {\calyear};
    \end{tikzpicture}
\end{document}
```

5. Compile the document and have a look:



2016

January

				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

February

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29						

March

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

April

				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

May

						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

June

		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

July

				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

August

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

September

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

October

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

November

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

December

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

How it works...

We used a matrix for easy positioning of the twelve months' calendars; we just specified the row and column spacing. While the given `calendar` style settings are pretty self-explanatory, modifying them requires knowledge of the options. The library is extensively explained in the TikZ manual, together with some examples, at `http://texdoc.net/pkg/tikz`.

Based on the date calculation features and existing styles of the calendar library, you can use the bells and whistles of TikZ to get a colorful, fancy calendar. Some outstanding examples can be found in the TikZ gallery at `http://www.texample.net/tikz/examples/feature/calendar-library/`.

Mimicking keys and menu items

Technical documentation and software manuals often require to tell the reader about keyboard shortcuts, how to use the program menu, or in which directory files can be located. This recipe helps with writing such explanations.

How to do it...

We will use the `menukeys` package and try out its main commands. Follow these steps:

1. Start a short document and load the `menukeys` package in it:

```
\documentclass[parskip=full]{scrartcl}
\usepackage{menukeys}
\begin{document}
\section*{Running \TeX works}
```

2. In the body text, use the `\menu` command for menu entries, the `\keys` command for keyboard combinations, and the `\directory` command for a path, as follows:

```
In the main menu, click \menu{Typeset > pdfLaTeX}
for choosing the \TeX\ compiler. Then press \keys{\cmd + T}
for typesetting. Click \menu{Window > Show > Fonts} for seeing
the fonts used by the document.

Press \keys{\shift + \cmd + F} for full screen.

The program is installed in
\directory{/Applications/TeX/TeXworks.app}.
\end{document}
```

3. Compile the document and take a look:

Running TEXworks

In the main menu, click Typeset ⟩ pdfLaTeX for choosing the TEX compiler. Then press ⌘ + T for typesetting. Click Window ⟩ Show ⟩ Fonts for seeing the fonts used by the document.

Press ⇧ + ⌘ + F for full screen.

The program is installed in `Applications‣TeX‣TeXworks.app`.

How it works...

The `menukeys` package provides three main commands which parse their arguments as a list:

- `\keys{combination}`: This command prints a key combination given by a list of keys separated by the `+` symbol
- `\menu{sequence}`: This command prints a sequence of menu entries separated by the `>` symbol
- `\directory{path}`: This command prints a file path in typewriter font, with components separated by the `/` symbol

The input as separated lists follows common conventions, and the output mimics what we expect to see on a computer screen.

Various aspects can be customized. There are several predefined styles, such as rounded and angular menus; rounded, angular and shadowed keys; and even vintage typewriter keys. Paths can use folder symbols. If this small example has whetted your appetite and you would like to modify the appearance further, have a look at the package manual. As usual, you can open it by typing the `texdoc menukeys` command in Command Prompt, or by visiting `http://texdoc.net/pkg/menukeys`.

6
Designing Tables

This chapter contains recipes for creating good-looking tables. Specifically, it includes recipes for how to do the following:

- Creating a legible table
- Merging cells
- Splitting a cell diagonally
- Adding footnotes to a table
- Aligning numeric data
- Coloring a table
- Adding shape, shading, and transparency
- Importing data from an external file

Introduction

With LaTeX, we can create and print complex tables. In this chapter, we will first talk about achieving good readability. Some recipes will show and explain useful design elements.

While the recipes are based on standard LaTeX `tabular` environments, you can use them in a similar manner with `tabular*`, `tabularx`, `tabulary`, and related environments.



The `tabular*` environment achieves a desired table width by adjusting the spacing between columns. The `tabularx` environment also spans to a given width by automatically calculating column widths, trying to distribute available space evenly. The `tabulary` environment also balances column widths, but tries to give more space to columns with more content. These environments and packages are covered in LaTeX introductions such as in the *LaTeX Beginner's Guide* written by me.

Even with the `array` environment, it's quite the same; it would just be in math mode. That's why you may also have a look at the recipes in *Chapter 10*, *Advanced Mathematics*, which deal with arrays and matrices. For example, highlighting in matrices, shown in *Chapter 10*, *Advanced Mathematics*, can be used with the `tabular` environment as well.

Creating a legible table

When we learn to write tables with LaTeX, we get to know how to write in rows and columns, and how to draw lines between cells and borders around the table. Though, using all borders can result in such a table:

$*$	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9

Such a habit may result from using **WYSIWYG** (**What You See Is What You Get**) software such as Excel or Word to write tables. However, while such a grid is useful for *entering* data, it makes *reading* really difficult.

In good books, we can find more legible tables. Let's take a look at how to create one.

How to do it...

We will use the `booktabs` package, which has been written with good design in mind. Specifically, it enhances lines in tables. It focuses on horizontal lines with improved spacing and adjustable thickness.

For our example, we will sketch a table that shows the availability of certain structuring features in LaTeX's base classes. Here's how to do this:

1. Specify the class. You could simply use the `article` class for now:
   ```
   \documentclass{article}
   ```
2. Load the `booktabs` package:
   ```
   \usepackage{booktabs}
   ```
3. Load the `bbding` package, which provides a check mark symbol:
   ```
   \usepackage{bbding}
   ```

4. Within the document body, set up a `table` environment and create the `tabular` layout within. For convenience, here's the complete remaining code to copy:

```
\begin{document}
\begin{table}
  \centering
  \renewcommand{\arraystretch}{1.6}
  \begin{tabular}{lccccc}
    \toprule
    Class   & Part page  & Chapters   & Abstract   &
                  Front-/Backmatter & Appendix name \\
    \cmidrule(r){1-1}\cmidrule(lr){2-2}\cmidrule(lr){3-3}
    \cmidrule(lr){4-4}\cmidrule(lr){5-5}\cmidrule(l){6-6}
    article &            &            & \Checkmark &  \\
    book    & \Checkmark & \Checkmark &            &
              \Checkmark & \Checkmark                 \\
    report  & \Checkmark & \Checkmark & \Checkmark &
                         & \Checkmark                 \\
    \bottomrule
  \end{tabular}
  \caption{Structuring differences between standard
           \LaTeX\ classes}
  \label{comparison}
\end{table}
\end{document}
```

5. Compile the document and inspect the result:

Class	Part page	Chapters	Abstract	Front-/Backmatter	Appendix name
article			✓		
book	✓	✓		✓	✓
report	✓	✓	✓		✓

Table 1: Structuring differences between standard LaTeX classes

How it works...

We used a `table` environment to get a table with a caption. This is the common container. From now on, we will focus on the actual structure. This is built and printed by the `tabular` environment. Both are well explained in LaTeX introductions, such as in the *LaTeX Beginner's Guide* by *Packt Publishing*. Refer to such a book to learn the details. Some are mentioned in the next recipe.

The basic syntax is this: an ampersand (`&`) ends a column, and a double backslash (`\\`) ends a row.



It's a good idea to end a line in the source code after `\\`. Also align ampersands, if possible, as this can improve code readability and make it easier to fill the right columns. You know that several consecutive spaces are treated as one, so some additional spacing doesn't hurt.

The `booktabs` package provides commands for horizontal lines, also called **rules**. This term originates from British typesetting. The differences between the `booktabs` package and standard LaTeX `tabular` lines are as follows:

- More space above and below a line by default
- Several kinds of rules with customizable thickness

Specifically, the commands are as follows:

- `\toprule`: This command is used for a line at the top, often made thicker
- `\midrule`: This command is used for lines within the table, commonly thinner than outer lines
- `\bottomrule`: This command is used for a line at the bottom, mostly with the same thickness as the top line
- `\cmidrule`: This command is used for a line spanning one, two, or more columns

The last command has a more sophisticated syntax:

```
\cmidrule[thickness](trim){a-b}
```

The `thickness` parameter is optional and can specify a desired thickness, such as `1pt`.

The `trim` parameter is also optional and specifies a horizontal trimming, `r` for right and `l` for left. Both can be combined, as we did in the preceding example.

The mandatory argument `a-b` defines that the line will span from column `a` to column `b`.

All commands understand an optional argument for the thickness; for example, you could also write the `\toprule[1pt]` command.

While the basic use can be seen in our example, the package manual explains more options for tweaking, such as changing the default thickness for each kind of rule and trim. You can read the manual by typing `texdoc booktabs` in Command Prompt, or by visiting `http://texdoc.net/pkg/booktabs`.

There's more...

Besides good tools such as `booktabs`, a well thought out design is key to great tables. So let's sum up good advice given by typographers:

- Respect the reading direction. This means the following:
  - Write text horizontally
  - Never use vertical lines
  - A few horizontal lines can support reading in the structure, such as below the header or between groups of content

  The last two rules can both be changed for tables, which really should be read vertically.
- Minimize what doesn't belong to the actual information, such as lines, dividers, boxes, colors, and font changes. Consequently, don't use double lines.
- Leave whitespace around the table. Separating lines additionally may not be necessary then.

Generally, whitespace gives invisible support to the structure; that's why we increased the default `\arraystretch` value from 1 to 1.6. In the next recipe, structuring by whitespace is even more visible.

Here's the final advice regarding the content:

- Align text for optimal readability or overall table design
- Align decimal values at the decimal point for better comparing
- Instead of repeatedly using units in cells, put the unit into the column header
- Instead of repeating values, consider merging neighboring cells

The recipes later in this chapter will support us regarding these points.

Adding footnotes to a table

Entries in tables, such as row header text, should be short, or they can present difficulties. For example, long headers could make it harder for our eyes to follow a row with short entries but wide skips. One approach to adding the necessary details while keeping the table itself short and crisp is using footnotes.

Instead of placing the notes at the foot of the page, it's a good idea to add the notes directly at the foot of the table. We also call them **table notes**. Some benefits of this are as follows:

- Tables are usually self-contained objects for reference.
- While common footnotes are written at the page bottom to retain the text flow, text following a table is usually independent of that table. So there's no need to defer notes further down.
- Tables can automatically move for better page breaks; their notes should stay together with the corresponding table.
- Table notes can be independent of text footnotes. This saves headaches about keeping numbering in order.

Footnotes in `minipage` environments already work this way, so you could wrap your table in a `minipage` environment as a first possibility. In this recipe, we will take a more sophisticated approach.

How to do it...

We will use the `threeparttable` package, which can generate footnotes below tables with the same width as the table body. With a normal `tabular` or related environment, which may be inside a `table` environment, follow these instructions:

1. Load the `threeparttable` package in your document preamble:

   ```
   \usepackage{threeparttable}
   ```

2. Surround your original `tabular` environment with a `threeparttable` environment. In other words, place the `\begin{threeparttable}` command before the `\begin{tabular}` command, and the `\end{threeparttable}` command after the command `\end{tabular}`.
3. Within a table cell, add table notes using the command `\tnote{symbol}`, where the symbol is mandatory and can be a number, letter, or symbol chosen by you. For example:

   ```
   … & cell text\tnote{1} & …
   ```

4. Right before the end of the `threeparttable` environment, which means right before the `\end{threeparttable}` command, insert a `tablenotes` environment. There, insert your footnotes in the form of a list:

```
\begin{tablenotes}
  \item[1] Your first remark
  \item[2] Another remark
\end{tablenotes}
```

Let's apply this to the first recipe of this chapter:

1. Take the full recipe code and insert the commands as just described. For our example, we will add three notes. The code now becomes as follows:

```
\documentclass{article}
\usepackage{booktabs}
\usepackage{bbding}
\usepackage{threeparttable}
\begin{document}
\begin{table}
  \centering
  \renewcommand{\arraystretch}{1.6}
  \begin{threeparttable}
    \begin{tabular}{lccccc}
      \toprule
      Class   & Part page  & Chapters   & Abstract\tnote{1}
        & Front-/Backmatter\tnote{2}
        & Appendix name\tnote{3} \\
      \cmidrule(r){1-1}\cmidrule(lr){2-2}\cmidrule(lr){3-3}
      \cmidrule(lr){4-4}\cmidrule(lr){5-5}\cmidrule(l){6-6}
      article &            &            & \Checkmark &  \\
      book    & \Checkmark & \Checkmark &            &
                \Checkmark & \Checkmark                 \\
      report  & \Checkmark & \Checkmark & \Checkmark &
                           & \Checkmark                 \\
      \bottomrule
    \end{tabular}
    \begin{tablenotes}
```

```
        \item[1] An environment: \verb|\begin{abstract}|
                 \ldots \verb|\end{abstract}|
        \item[2] Commands: \verb|\frontmatter|,
                 \verb|\mainmatter|, \verb|\backmatter|
        \item[3] The command \verb|\appendix| exists in
                 Article too, but there's no prefix
                 ``Appendix''.
      \end{tablenotes}
    \end{threeparttable}
    \caption{Structuring differences between standard
             \LaTeX\ classes}
    \label{comparison}
  \end{table}
  \end{document}
```

2. Compile that code and examine the changes:

Class	Part page	Chapters	Abstract[1]	Front-/Backmatter[2]	Appendix name[3]
article			✓		
book	✓	✓		✓	✓
report	✓	✓	✓		✓

[1] An environment: `\begin{abstract}` ... `\end{abstract}`
[2] Commands: `\frontmatter`, `\mainmatter`, `\backmatter`
[3] The command `\appendix` exists in article too, but there's no prefix "Appendix".

Table 1: Structuring differences between standard LaTeX classes

How it works...

In contrast to the classic `\footnote` command, we have to do some manual work in addition, as follows:

- Choose identifiers
- Take care of the order ourselves
- Write the list of notes manually

Like a `tabular` environment, a `threeparttable` environment takes an optional argument for vertical placement, which can be `t` for top, `b` for bottom, or `c` for centered alignment. Top alignment is the default. It also doesn't float. However, you can put it into a `table` environment as usual, with a caption and label for referencing, so it is able to float.

The behavior of `threeparttable` notes can be customized by options. They can be globally given to the `\usepackage` command, or locally applied to a `tablenotes` environment. These are as follows:

- `para`: Table notes will subsequently be printed, without line breaks between them
- `flushleft`: There will be no hanging indentation of table notes
- `online`: Instead of superscript, table note symbols will be printed by the `\item` command in normal size at the line base
- `normal`: This is the default formatting, which means superscript symbols, hanging indentation, and line breaks between table notes

Further commands for fine-tuning are provided and are described in the package manual. You can open it by typing the `texdoc threeparttable` command in Command Prompt. You can also access it on the Internet at `http://texdoc.net/pkg/threeparttable`.

Aligning numeric data

Standard alignment options in table columns are left, right, and centered. In cases of numeric values, this might not be sufficient. The only way to keep number magnitudes comparable is aligning digits at certain positions, such as at the ones, thousands, or millions places, or at decimal points. Integers can simply be right aligned. Numbers with decimal fractions could be filled up with zeroes to get decimal points aligned, but that would add vacuous noise.

In the case of fractions, it's good to directly align at the decimal points. In this recipe, we will implement this.

How to do it...

The `siunitx` package is, first of all, intended for typesetting values with units consistently. Incidentally, supporting its primary purpose, it provides a `tabular` column type for aligning at decimal points. We will use this now:

1. Load the `siunitx` package in your preamble:

   ```
   \usepackage{siunitx}
   ```

2. Use `S` as the column specifier for a column with alignment at decimal points, such as:

   ```
   \begin{tabular}{lSS}
   ```

3. Within a table cell, simply write the number. Also, it's good to insert some spaces in the source code to get decimal points aligned for legibility. Repeating space characters doesn't hurt, but it's not required.
4. To avoid this special alignment, such as in the case of row headers, enclose the cell text in curly braces, like so:

   ```
   … & {atomic mass} & …
   ```

Let's try it with a very short complete example. We will use the `siunitx` package as described just now, and the `chemformula` package in addition. The `chemformula` package makes typing chemical formulas easier. You will see more of this package in *Chapter 11, Science and Technology*. Follow these steps:

1. Put this code into your LaTeX editor:

```
\documentclass{article}
\usepackage{booktabs}
\usepackage{siunitx}
\usepackage{chemformula}
\begin{document}
  \begin{tabular}{lSS}
    \toprule
              & {atomic mass} & {total mass} \\
    \midrule
    \ch{C}    &    12.011     & 12.011       \\
    \ch{H}    &     1.00794   &  6.04764     \\
    \ch{C2H6} &               & 30.06964     \\
    \bottomrule
  \end{tabular}
\end{document}
```

2. Compile the example, and take a look:

	atomic mass	total mass
C	12.011	12.011
H	1.007 94	6.047 64
C_2H_6		30.069 64

How it works...

By default, a `S` column places the numbers in such a way that the decimal points are in the center of the cell and horizontally aligned to each other.

You can customize the alignment of numbers in a `S` column. Proper alignment implies reserving space for the numbers. You can specify the number of integers and decimal places and the alignment type by using the `\sisetup` command of the `siunitx` package, as follows:

```
\sisetup{table-format = 2.5,
    table-number-alignment = right}
```

Here, the `table-format` parameter will be parsed so that the `siunitx` package knows that we reserve space for two integer figures and five decimal places.

The format string can be even more sophisticated:

```
table-format = +2.5e+2
```

This additionally reserves space for an exponent with two decimals, an exponent sign, and a mantissa sign. This is useful for numbers such as $-22.31442 \times 10^{-10}$.

The package manual describes further options. It is worth reading, since the `siunitx` package is an excellent choice for when you need to print numbers with units. You can read the manual by typing the `texdoc siunitx` command in Command Prompt, or you can see it on the Internet at `http://texdoc.net/pkg/siunitx`.

You will encounter this package again in *Chapter 11*, *Science and Technology*.

There's more...

There are further packages that can be used for the same purpose, namely `dcolumn` and `rccol`. Both provide `tabular` column types for proper alignment of numbers. They are actually the classics, but the `siunitx` package is a capable, innovative, and very actively maintained package that even impelled LaTeX3 development, so you can safely use it for future work.

Coloring a table

Sometimes we see "zebra striped" tables; tables with alternating row colors. This design is intended to support horizontal reading without the need for separating lines.

While some people like the design, some say it may be harder to read. For example, while looking at a table, the eye may scan at first one color, and then need to jump back, scanning the rows with the other color. So, when we decide to go for this design, we should consider doing the following:

- Make the color variation small to prevent the tendency of the eye to jump across even or odd rows. The information should be visually stronger than the distinction of those two layers.
- Keep a good contrast between color and text. For example, black text with a dark gray background is hardly readable.
- Have a different color for the header in order to emphasize it.

How to do it...

We will use the `xcolor` package for this task. The sample data for this table, which we will use in later recipes as well, has been taken from `DistroWatch.com`.

Let's work on an example:

1. Start with a `document` class:

```
\documentclass{article}
```

2. Load the `xcolor` package with the `tables` option for table support:

```
\usepackage[table]{xcolor}
```

3. Declare alternating row colors:

```
\rowcolors{2}{gray!15}{white}
```

4. Define a macro for the table header appearance:

```
\newcommand{\head}[1]{%
  \textcolor{white}{\textbf{#1}}}
```

5. Enlarge the default `tabular` line spacing:

```
\renewcommand{\arraystretch}{1.5}
```

6. In the actual `tabular` environment, use the command `\rowcolor{color}` to color any row, and the `\head{text}` command for cells with header commands. Here is the code for the table within the remaining document body:

```
\begin{document}
\begin{table}[ht]
  \centering
  \sffamily
  \begin{tabular}{rlr}
    \rowcolor{black!75}
     & \head{Distribution} & \head{Hits} \\
    1 & Mint        & 2364 \\
    2 & Ubuntu      & 1838 \\
    3 & Debian      & 1582 \\
    4 & openSUSE    & 1334 \\
    5 & Fedora      & 1262 \\
    6 & Mageia      & 1219 \\
    7 & CentOS      & 1171 \\
    8 & Arch        & 1040 \\
    9 & elementary &  899 \\
   10 & Zorin       &  851 \\
 \end{tabular}
\end{table}
\end{document}
```

The following is the output of the preceding code:

	Distribution	Hits
1	Mint	2364
2	Ubuntu	1838
3	Debian	1582
4	openSUSE	1334
5	Fedora	1262
6	Mageia	1219
7	CentOS	1171
8	Arch	1040
9	elementary	899
10	Zorin	851

How it works...

The `xcolor` package implicitly loads the `colortbl` package, which is the standard package for coloring tables. The `colortbl` package provides three main commands:

- The command `\columncolor[color model]{color name}[left overhang][right overhang]` with a `xcolor` model such as `rgb` or `cmyk`, a predefined color name, and color overlap to the left and to the right. Only the color name is mandatory. The `\columncolor` command is intended for placing within `tabular` column definitions by `>{...}`.
- The `\rowcolor` command with the same arguments. Left and right arguments are practically useless. The command has to be at the beginning of the first cell in the row and is valid for the whole row.
- The `\cellcolor[color model]{color name}` command for a single cell.

Using these commands, you can color tables in any way you want.

The `xcolor` package provides another useful command, which we used in our example:

```
\rowcolors[commands]{start row}{odd-row color}{even-row color}
```

This one colors odd and even rows with alternating colors, starting from a given row. Optionally, you can supply commands executed at each row. Examples of such commonly-used commands are `\hline` and `\noalign{...}`.

In our example, we used just one `\rowcolor` command for the header, and one `\rowcolors` command for the remaining rows. For better readability within a head row with a darker color, we used white for the text color. Alternatively, with the black text color, you could choose a color which keeps the text readable, such as `\rowcolor{gray!35}`.

Both the `colortbl` and `xcolor` package manuals explain further commands and details, accessible via `texdoc` and `http://texdoc.net`, like you saw with other packages in the previous sections.

Merging cells

As indicated in the table design advice of the last recipe, instead of repeating identical values in adjacent cells, you could leave the other cells empty if the reader would be able to understand that the same values apply.

We can support the meaning by merging cells and centering the cell value over the whole new width.

How to do it...

Merging and centering can be done horizontally, vertically, or both combined. We will start with the horizontal method, spanning cells over multiple columns. You can see this often, with table headers that apply to several columns. So, in this recipe, we will conflate header texts.

As modeling clay, we will take the differences between various LaTeX compilers. While LaTeX remains the same, the underlying TeX engine causes differences. We will arrange them now, using these steps:

1. Specify the class. You could simply use the `article` class for now:
   ```
   \documentclass{article}
   ```
2. Load the `array` package, which provides useful commands:
   ```
   \usepackage{array}
   ```
3. Load the `booktabs` package to get nicer lines:
   ```
   \usepackage{booktabs}
   ```
4. Load the `metalogo` package to write TeX logos:
   ```
   \usepackage{metalogo}
   ```
5. Define how you would like to stretch the table row spacing:
   ```
   \renewcommand{\arraystretch}{1.6}
   ```
6. To get the table, copy the following code into your editor; an explanation follows in the next section. For completeness, this is the whole remaining document body:
   ```
   \begin{document}
   \begin{tabular}{@{}p{1.5cm}p{1.6cm}>{\raggedleft}p{1cm}
                   >{\raggedright}p{1.6cm}r@{}}
     Compiler & \multicolumn{2}{c}{Input}
              & \multicolumn{2}{c@{}}{Output} \\
      \cmidrule(r){1-1}\cmidrule(lr){2-3}\cmidrule(l){4-5}
              & Encoding & Images  & Fonts & Format \\
      \cmidrule(lr){2-2}\cmidrule(lr){3-3}
      \cmidrule(lr){4-4}\cmidrule(l){5-5}
      \LaTeX & utf8, ascii, applemac, latin1, \ldots
        & EPS & Type 1, Type 3 & DVI \\
      pdf\LaTeX & utf8, ascii, applemac, latin1, \ldots
       & PDF PNG JPG & Type 1, Type 3 & PDF \\
   ```

```
  \XeLaTeX, \LuaLaTeX & utf8 & PDF PNG JPG & Type 1,
          Type 3, OpenType, Graphite, TrueType & PDF \\
\end{tabular}
\end{document}
```

7. Compile the document and take a look at the result:

Compiler	Input		Output	
	Encoding	Images	Fonts	Format
LaTeX	utf8, ascii, applemac, latin1, ...	EPS	Type 1, Type 3	DVI
pdfLaTeX	utf8, ascii, applemac, latin1, ...	PDF PNG JPG	Type 1, Type 3	PDF
XeLaTeX, LuaLaTeX	utf8	PDF PNG JPG	Type 1, Type 3, OpenType, Graphite, TrueType	PDF

How it works...

We will take a minute to look at the `tabular` options:

- `@{code}`: This inserts the given code instead of the column separation space. The `code` can be a command such as a symbol or a space macro. Here, we set it to be empty, thus achieving left alignment with the lines.
- `p{width}`: This specifies cells of a certain width with wrapping text. The `p` character stands for *paragraph*.
- `>{code}`: This inserts code before a cell of the column, while `<{code}` does it afterwards. This comes from the `array` package and is documented in the manual. Here, we use it to insert the `\raggedleft` command to avoid the default full justification, which isn't so nice in table cells.
- `r`, `l`, and `c`: These stand for right aligned, left aligned, and centered, respectively.

Now, look at the `Input` cell, which spans two columns. We used the following command:

```
\multicolumn{number of columns}{formatting}{text}
```

Here, we set 2 as the number of columns to span, `c` for centered formatting, and finally the cell text follows.



Besides actually merging cells, the `\multicolumn` command is a simple way of changing the formatting of a single cell: the `\multicolumn{1}{formatting}{text}` command defines the formatting of a single cell independently of the table global column formatting.

For this table, we used very few lines to indicate the structure, which is nevertheless well visible thanks to the spacing.

There's more...

As mentioned, we can also span cells over multiple rows. If there's an odd number of rows to span, we can simply fill the row in the middle. So, let's take a look at a case in which we need centering by merging.

Compile this short example showing selected classes of major LaTeX bundles:

```
\documentclass{article}
\usepackage{booktabs}
\usepackage{multirow}
\begin{document}
\begin{tabular}{cc}
  Bundle & Main classes \\
  \cmidrule(lr){1-1}\cmidrule(lr){2-2}
  \addlinespace
  \multirow{4}{*}{\LaTeX\ base} & article  \\
       & book     \\
       & report   \\
       & letter   \\
  \addlinespace
  \multirow{4}{*}{KOMA-Script}  & scrartcl \\
       & scrbook  \\
       & scrreprt \\
       & scrlttr2 \\
\end{tabular}
\end{document}
```

This code will produce the following result:

Bundle	Main classes
LaTeX base	article book report letter
KOMA-Script	scrartcl scrbook scrreprt scrlttr2

We loaded the `multirow` package and used its command:

```
\multirow{number of rows}[struts]{width}[correction value]{text}
```

Here, the value of the `number of rows` argument may even be negative, in which case the spanning would reach backwards. The optional `correction value` argument can be a positive or negative LaTeX length added for fine-tuning, if given.

The other optional argument, `struts`, is only of interest if you inserted so called `bigstruts`, which are helping lines for stretching rows; here, you may specify their number. For details, refer to the `multirow` manual. Open it by typing the `texdoc multirow` command in Command Prompt, or at `http://texdoc.net/pkg/multirow`.

While you can define a `width` value for the text, we wrote a `*` symbol to use the natural text width.

Splitting a cell diagonally

If we need a header for the first column but also for the entries of the first row, the top-left cell can be split to contain both header entries, separated by a diagonal line.

How to do it...

We will use the `slashbox` package. It is part of **MiKTeX**, but not of **TeX Live**. Users of TeX Live can download it from **CTAN** at `http://ctan.org/pkg/slashbox`.

In this recipe, we will build a time table. It's intended to be filled out by hand later, so we will use vertical lines for delimiting fields. Follow these steps:

1. Use any document class. Here, we simply use the `article` class:

```
\documentclass{article}
```

2. Load the `slashbox` package:

```
\usepackage{slashbox}
```

3. Within the document body, create the `tabular` layout:

```
\begin{document}
\renewcommand{\arraystretch}{1.8}
\begin{tabular}{|l|c|c|c|c|c|}
  \hline
  \backslashbox{Time}{Weekday} & Monday   & Tuesday
                & Wednesday    & Thursday & Friday \\
  \hline
  8--10  & & & & & \\
  10--12 & & & & & \\
  12--14 & & & & & \\
  14--16 & & & & & \\
  \hline
\end{tabular}
\end{document}
```

4. Compile and take a look:

Time / Weekday	Monday	Tuesday	Wednesday	Thursday	Friday
8–10					
10–12					
12–14					
14–16					

How it works...

We used vertical lines, because the table should be read by day, that is, by column. The `slashbox` package provides two commands:

- `\slashbox[width][trim]{left top text}{right bottom text}`
- `\backslashbox[width][trim]{left bottom text}{right top text}`

The optional arguments are for adjustments in cases when the automatic calculation doesn't fit. You can specify the width of the slashed column. You can choose the trimming of the default left and right column separation space by stating `l`, `r`, or `lr`, for left trim, right trim, or cutting at both sides, respectively.

The diagonal line will be from the upper left to the lower right corner, like a backslash, or from the lower left to the upper right corner, like a slash symbol.

Adding shape, shading, and transparency

While tables in books and theses are expected to be sober, presentations at talks are often more colorful. In this recipe, we will take a look at how to add graphical elements, such as rounded borders and shaded colors, to tables.

How to do it...

We will use the `beamer` class, which is very popular for doing presentations. We will take an ordinary `tabular` environment, put it into a TikZ node, and add shape and color using TikZ.

It's a challenging example, but it shows what we can achieve. Follow these steps:

1. Choose the `beamer` class, remove the `navigation symbols` argument, and add a background template with some shading:

```
\documentclass{beamer}
\setbeamertemplate{navigation symbols}{}
\setbeamertemplate{background canvas}[vertical shading]%
  [top = blue!1, bottom = blue!40]
```

2. Load the `booktabs` package for nicer lines:

```
\usepackage{booktabs}
```

3. Load TikZ and its `calc` library, and add a background layer:

```
\usepackage{tikz}
\usetikzlibrary{calc}
\pgfdeclarelayer{background}
\pgfsetlayers{background,main}
```

4. Define how you would like to stretch the table row spacing:

```
\renewcommand{\arraystretch}{1.6}
```

5. Define some macros for symbols to use:

```
\newcommand{\up}{\textcolor{green}{$\blacktriangle$}}
\newcommand{\down}{\textcolor{red}{$\blacktriangledown$}}
\newcommand{\same}{\textcolor{darkgray}{\textbf{--}}}
```

6. Use another macro to print the heading, since we don't have visible sections:

```
\newcommand{\heading}[1]{{\Large\bfseries #1\par\medskip}}
```

7. Start a document body, a frame with centering, and the heading, and begin a TikZ picture:

```
\begin{document}
\begin{frame}
  \centering
  \heading{Linux distribution ranking, December 14, 2014}
  \begin{tikzpicture}
```

8. Declare a node, choose the name table, and put the whole `tabular` environment into it:

```
    \node (table) {
      \begin{tabular}{rlrcc}
        & \textbf{Distribution} & \textbf{Hits} & \\
        \addlinespace[2pt]
        1 &  Mint      & 2364 & \down \\
        \midrule
        2 & Ubuntu     & 1838 & \up   \\
        \midrule
        3 & Debian     & 1582 & \same \\
        \midrule
```

```
    4 & openSUSE   & 1334 & \up   \\
    \midrule
    5 &  Fedora    & 1262 & \up   \\
    \midrule
    6 & Mageia     & 1219 & \down \\
    \midrule
    7 & CentOS     & 1171 & \same \\
    \midrule
    8 & Arch       & 1040 & \same \\
    \midrule
    9 & elementary &  899 & \same \\
    \midrule
    10 & Zorin     &  851 & \down \\[0.5ex]
  \end{tabular}};
```

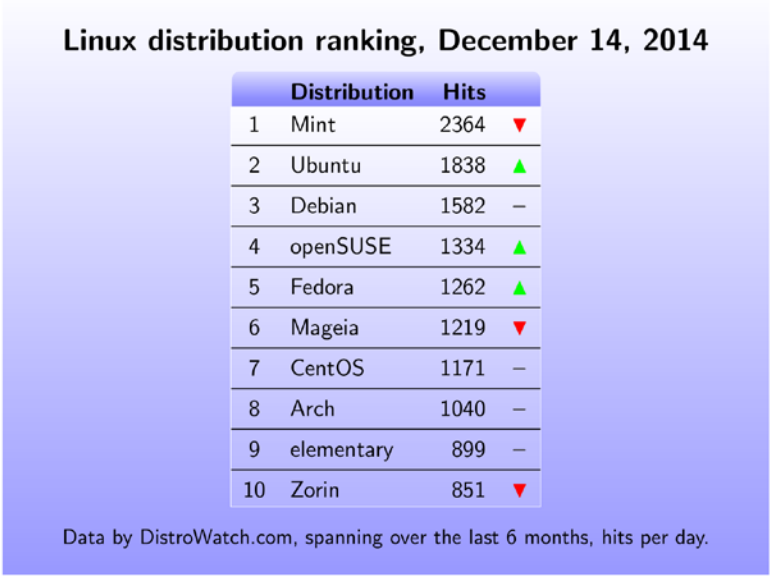
9. Draw a colored header and body onto the background layer so it will actually be put below the text:

```
\begin{pgfonlayer}{background}
  \draw[rounded corners, top color=blue!20,
    bottom color=blue!80!black, middle color=blue!80,
      opacity=0.5, draw=white]
      ($(table.north west)+(0.14,0)$)
    rectangle ($(table.north east)-(0.13,0.9)$);
  \draw[top color=blue!1,bottom color=blue!30,
    draw=white] ($(table.north east)-(0.13,0.6)$)
    rectangle ($(table.south west)+(0.13,0.2)$);
\end{pgfonlayer}
```

10. End the TikZ picture, add some explanatory text, and close the frame and document:

```
  \end{tikzpicture}
  \small
  Data by DistroWatch.com, spanning over the last 6 months,
  hits per day.
\end{frame}
\end{document}
```

11. Compile the presentation and take a look:



How it works...

The steps are straightforward, and the drawing syntax is explained in the TikZ manual. It is too much to explain TikZ in depth here, so please refer to the manual at `http://texdoc.net/pkg/tikz`, or by typing the `texdoc tikz` command in the command line.

You will see much more of TikZ in *Chapter 9*, *Creating Graphics*. For now, let's just mention some important points:

- The `beamer` package is responsible for the basic background, which we shaded in light blue. Normally, you would choose a beamer theme with a predesigned layout.
- TikZ is an extremely capable graphics package. Once we start using it, the possibilities are endless.
- TikZ is modular. We loaded its `calc` library because it provides a syntax that we used to add coordinates, as in `($(table.north west)+(0.14,0)$)`.
- The `\midrule` command is a better separation line provided by the `booktabs` package. You could change the line color by using the `colortbl` package:

```
\usepackage{colortbl}
\arrayrulecolor{blue}
```

The `pgf` commands come from the pgf/TikZ bundle, where **PGF** stands for **portable graphics format**. The `pgf` package is the system layer below the user front-end TikZ. We used it to draw on a background layer:

- We drew a rectangle with rounded corners on the background and a rectangle as a background for the table body, which actually overlaps the table, so we don't need to take care of the lower rounded corners.
- We used a lot of colors and TikZ did the transition between top and bottom colors. The syntax `color1!percent!color2` produces a mix of `color1` (its amount stated in percent) and `color2`.

There's more...

We can automate the numbering of the table rows. Another approach would be to create a table graphically built with nodes. Let's have a look at these options.

Numbering table rows automatically

This automatism makes swapping rows much easier if numbers of hits, and thus ranks, change. Furthermore, it can prevent mistakes.

You can achieve the same resulting table as the previous one by following these steps:

- Load the `array` package in addition for extended `tabular` syntax:

```
\usepackage{array}
```

- Define a counter and a macro which raises it by one and prints it:

```
\newcounter{rank}
\setcounter{rank}{0}
\newcommand{\steprank}{\stepcounter{rank}
  \makebox[2em]{\therank}\hspace{\tabcolsep}}
```

- The new macro will be inserted into the `tabular` column definition, and takes effect in each row. Only the first row is differently formatted, to prevent numbering of the header. The table itself becomes simpler now:

```
\begin{tabular}{@{\steprank}lrcc}
\multicolumn{1}{l}{\hspace{2em}\textbf{Distribution}} &
\textbf{Hits} & \\
\addlinespace[2pt]
Mint & 2364 & \down \\
\midrule
Ubuntu & 1838 & \up \\
…
```

Adding styles to rows and columns

You can take a very different approach. Instead of using a standard `tabular` environment, you could change to a TikZ matrix. TikZ provides a library for building and printing matrices with the styles and capabilities of graphics packages. The advantage is that you can natively apply style definitions and get all cells as nodes, which you can refer to later, for things like highlighting and drawing arrows.

We can show you a sample that illustrates such capabilities. The explanations here and later will help. For specific details, you may have a look into the TikZ or `beamer` manuals. Here it goes:

```
\documentclass{beamer}
\usepackage{tikz}
\usetikzlibrary{matrix}
\setbeamertemplate{background canvas}[vertical shading]%
  [top=blue!1,bottom=blue!40]
\setbeamertemplate{navigation symbols}{}
\newcommand{\up}{\textcolor{green!75!black}{$\blacktriangle$}}
\newcommand{\down}{\textcolor{red}{$\blacktriangledown$}}
\newcommand{\same}{\textcolor{darkgray}{\textbf{--}}}
\newcounter{rank}
\setcounter{rank}{0}
\newcommand{\steprank}{\stepcounter{rank}\therank}
\newenvironment{matrixtable}[4]{%
  \begin{tikzpicture}
  \matrix (m) [matrix of nodes, nodes in empty cells,
    nodes={draw, top color=blue!10, bottom color=blue!35,
      inner sep=2pt, minimum height=3.5ex, anchor=center},
    top color=blue!20, bottom color=blue!80, draw=white,
    column sep=1ex, row sep=0.6ex, inner sep=2ex,
    rounded corners,
    row 1/.style    = {font=\bfseries},
    column 1/.style = {minimum width=#1, font=\steprank},
    column 2/.style = {minimum width=#2},
    column 3/.style = {minimum width=#3},
    column 4/.style = {minimum width=#4}]}%
  {;\end{tikzpicture}}
\begin{document}
```

```
\begin{frame}[fragile]
  \begin{center}
    \begin{matrixtable}{1.2cm}{2.4cm}{1.2cm}{0.6cm}{
      Rank & Distribution & Hits & ?      \\
           & Mint         & 2364 & \down  \\
           & Ubuntu       & 1838 & \up    \\
           & Debian       & 1582 & \same  \\
           & openSUSE     & 1334 & \up    \\
           & Fedora       & 1262 & \up    \\
           & Mageia       & 1219 & \down  \\
           & CentOS       & 1171 & \same  \\
           & Arch         & 1040 & \same  \\
           & elementary   &  899 & \same  \\
           & Zorin        &  851 & \down  \\}
    \end{matrixtable}
  \end{center}
\end{frame}
\end{document}
```

Compile it, and you will get this picture:



For convenience, we made an environment for the table with settings, so we don't need to repeat the style settings for every single table. Instead, we can have a consistent style throughout the document. Some width parameters give us further flexibility.

The lengthy but elaborate TikZ syntax is pretty self-explanatory for styling such as color, separations, width, and height, at least for reading. To extend it, reading the section about the `matrix` library in the manual is recommended, and in consequence, reading about nodes and styles there as well.

Like previously in this recipe, we let LaTeX count and print the rank number. We used a small trick: putting our `\steprank` command, which increments and displays the rank, into the font style for the first column. This hook was originally intended for applying a color to each cell of the first column; we were so daring to use it for calculating and printing.

We can easily modify single cells using TikZ syntax. Each cell is a node, and if we place something between bars before the cell content, it will be applied to that node. For example, writing the `|[red]|Fedora` argument in this `matrixtable` environment gives us Fedora written in red.

Each cell of our matrix is a node object and can be addressed using the syntax `(m-row-column)`. So, if we want to add the option `[remember picture]` to the `tikzpicture` argument in our `matrixtable` environment definition, we can use those node names. For example, we can draw an arrow from Mint to Debian by placing this code after the table, still in the same frame:

```
\begin{tikzpicture}[remember picture,overlay]
  \draw [->] (m-2-2) -- (m-4-2);
\end{tikzpicture}
```

That's just a glimpse of what we can do once we dress the table in TikZ.

Importing data from files

It can be very handy to get the data for the row entries from an external file. Actually, this can be done in a few lines.

How to do it...

We will load the `datatool` package, let it import data from a comma separated data file, sort it, and print it:

1. Store your data in the same folder as your main `tex` document. Here, we will use the data from the last recipes, unsorted within a text file:

```
Distribution,Hits
Zorin,85
Debian,1582
CentOS,1171
elementary,899
Arch,1040
Mint,2364
openSUSE,1334
Ubuntu,1838
Mageia,1219
Fedora,1262
```

2. Start with a document class, such as `article`, and load the `booktabs` package for improved `tabular` layout, and the `datatool` package for data handling:

```
\documentclass{article}
\usepackage{booktabs}
\usepackage{datatool}
```

3. With these `datatool` commands, load the data from the file and sort it:

```
\DTLloaddb{Linux}{linux.csv}
\DTLsort{Hits=descending}{Linux}
```

4. Start the document and set up a `tabular` environment:

```
\begin{document}
\begin{tabular}{rlr}
  & Distribution & Hits \\
  \cmidrule(lr){2-2}\cmidrule(lr){3-3}
```

5. Use this command to iterate through the data, building `tabular` rows:

```
\DTLforeach{Linux}{%
  \distribution=Distribution, \hits=Hits}{%
    \theDTLrowi & \distribution & \hits \\}
```

6. End the `tabular` environment and the document:

```
\end{tabular}
\end{document}
```

7. Compile and take a look:

	Distribution	Hits
1	Mint	2364
2	Ubuntu	1838
3	Debian	1582
4	openSUSE	1334
5	Fedora	1262
6	Mageia	1219
7	CentOS	1171
8	Arch	1040
9	elementary	899
10	Zorin	85

How it works...

The `datatool` package is actually a very capable bundle of packages for working with external data. Here, we simply imported a file, sorted it by a chosen key, and printed it.

Let us review the steps:

- We loaded the `datatool` package using the `\usepackage` command.
- We loaded the file `linux.csv` into a database with the name Linux. The command syntax is as follows:

  ```
  \DTLloaddb[options]{database name}{file name}
  ```

  Here, `options` may contain the following:

  - `omitlines=n`: This specifies an integer number `n` of lines to skip at the start.
  - `noheader=true` or `noheader=false`: This indicates whether or not the file contains a header. The default behavior is to assume a header. If you specify `noheader` without a value, `true` is assumed.

- `keys={key1,key2, ...}`: This specifies the keys for the database; it would override an existing header.
- `header={header1,header2, ...}`: This specifies the headers for the database and would override a header in the file. These entries would be used, for example, in easy printing with `\DTLdisplaydb{database name}`.

By default, it's assumed that data is separated by commas. You can change it with the `\DTLsetseparator{character}` command to any character, or by calling the `\DTLsettabseparator` command without argument to choose the tab as separator.

- We sorted the database with this command:

  ```
  \DTLsort[replacement keys]{sort criteria}{database name}
  ```

  Here, the `sort criteria` argument is a list of keys with optional order, which can be set to `descending` or `ascending`. The latter is the default if you just list keys. The optional list in the `replacement keys` argument is used, in the given order, if the current key is empty.

- While we could simply print all with the `\DTLdisplaydb{database name}` command, or the `\DTLdisplaylongdb{database name}` command using the `longtable` package, we iterated through the database for maximum flexibility:

  ```
  \DTLforeach[condition]{database name}{assign list}{text}
  ```

  The result of this macro is text for each row, which can include commands, but with replacements done with `assign list` in the form of a `command=key` list, as in our recipe. We did not use the optional `condition` argument, which can be a Boolean value calculated by the `\ifthenelse` command and related commands. This value, by default `true`, decides whether or not to apply `text` for the current row.

If this short sample convinced you of this method, you will find that the manual has extensive information and is worth reading. You can do a lot more. Similar to most packages, you can open the `datatool` manual by typing `texdoc datatool` in Command Prompt, or you can read it on the Internet at `http://texdoc.net/pkg/datatool`.

7
Contents, Indexes, and Bibliographies

In this chapter, we will cover the following recipes:

- Tuning the table of contents, lists of figures, and tables
- Creating a bibliography
- Adding a glossary
- Creating a list of acronyms
- Generating an index

Introduction

LaTeX provides features for the automatic creation of tables of contents, lists of figures, tables, bibliographies, glossaries, and indexes. This chapter contains recipes for quickly starting and customizing such lists.

We will start with the table of contents. While it's extremely easy to create a table of contents by simply using the `\tableofcontents` command, the format is quite rigidly determined by the document class. We will look at ways of modifying it with ease. The same can be used for the lists of figures and the lists of tables, which can be generated using the commands `\listoffigures` or `\listoftables`, respectively.

For a scientific work, especially, a list of references is important. It's also called a bibliography. One of the recipes in this chapter will provide a quick start using the very modern `biblatex` package.

Another recipe will show you how to create a glossary, so you can explain scientific or technical vocabulary to the reader.

The final recipe demonstrates generating a sorted index, which will let your reader look up key words easily.

Tuning a table of contents, lists of figures, and tables

When we start using LaTeX, the automatic creation of a table of contents (ToC) is impressive. The ToC with sectioning titles and page numbers is generated from your heading texts in commands such as `\part`, `\chapter`, and `\section`. The lists of figures and tables work in exactly the same way while they use the texts in the `\caption` command for the list entries. The design of the default table of contents follows a very common style. In some cases, though, you might need to adjust it.

How to do it...

We will use the `tocstyle` package to customize the table of contents. It belongs to the KOMA-Script bundle. You can try the settings of the recipe in your own document, or use the example from *Chapter 1*, *The Variety of Document Types*.

Automatic correction of widths

The default widths of numbers of sectioning numbers and page numbers in the ToC are static. This can lead to problems with large sectioning numbers, as well as with large page numbers. It's even more relevant for Roman numbering, where the width of numbers is even wider. In such cases, numbers and textual entries of the ToC may overlap.

You can fix this easily by using the following command:

```
\usepackage[tocindentauto]{tocstyle}
```

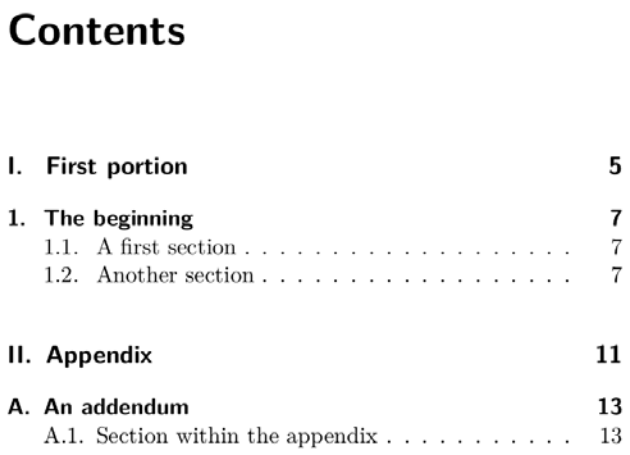
The name of this option, `tocindentauto`, may not look very intuitive. However, it means that the textual entries are automatically indented in such a way that they don't overlap with the numbers.



Note that producing a ToC now needs three runs, we will explain the reason later in this recipe.

Printing a flat table of contents

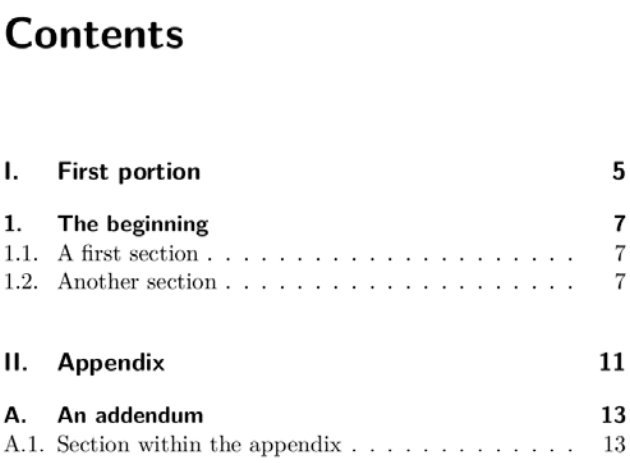
By default, a ToC is graduated. This means that sectioning entries of lower level headings will be indented. Our book example from *Chapter 1*, *The Variety of Document Types*, has a graduated table of contents:



To have all contents aligned to the left, specifically the sectioning numbers, call the following command:

```
\usepackage[tocflat]{tocstyle}
```

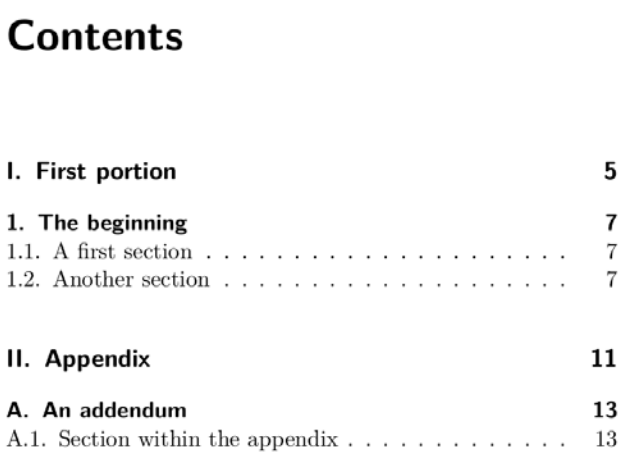
The ToC will now change to the following:



However, the entries will be aligned, which means that the sectioning numbers will be set in a box of equal width. You can omit this, pushing all the contents to the left, by using the following:

```
\usepackage[tocfullflat]{tocstyle}
```

Now you get the following:



The effects of the changes would be even more noticeable if subsections were added to the ToC.



You can combine options by writing them separated by commas, as follows: `\usepackage[tocindentauto,tocflat]{tocstyle}`.

Getting KOMA-Script like-sans-serif headings

KOMA-Script classes use a modern design of headings that are sans-serif and bold, and not as heavy as the default classic serif headings. Also, bold ToC entries are sans-serif, as you can see it the preceding output.

You can get the same behavior with other classes by using the following command:

```
\usepackage{tocstyle}
\usetocstyle{KOMAlike}
```

How it works...

To benefit from the automatic adjustments of **tocstyle**, you need several compiler runs. Expect at least three runs. In the first run, LaTeX writes all entries from commands, such as `\chapter` and `\section`, to the external table of contents file. The name of this file ends with `.toc`. For the list of figures and tables, the filenames end with `.lof` and `.lot`, respectively. In the following run, LaTeX can read the entries of the external file for printing the ToC. Widths are calculated and written to the `.aux` file. In the next run, the ToC is produced using the known entries from the `.toc` file and the known widths from the `.aux` file. Of course, changes in sectioning and page numbers would require a further run. This is a general consequence of straight compiling using external files: several compiler runs can be required until things are completely settled.

There's more...

If you load the `tocstyle` package without options, it produces a graduated ToC, just like the standard classes. However, it tries hard to avoid page breaks between ToC entries and their parents, and it automatically adjusts the widths. So, simply loading it can be a relief.

The package actually provides many commands for fine-tuning the style of the ToC, which is otherwise quite challenging. Refer to the package manual for details, which you can open by typing the `texdoc tocstyle` command in the Command Prompt, or by going to `http://texdoc.net/pkg/tocstyle`.

Creating a bibliography

The classic way of producing a bibliography, that is, a list of references, is using **BibTeX**. This is an external program for producing a bibliography from a plain text database with a chosen style, and for citing references in text.

In *LaTeX Beginner's Guide*, *Packt Publishing*, there's a tutorial for creating bibliographies using BibTeX. Refer to this guide to learn about the classic way. In our cookbook, we will use an advanced package called `biblatex`, which is generally considered to be the successor of BibTeX. It is a complete reimplementation of LaTeX's standard bibliographic features. It supports the BibTeX database format, but understands other formats. The `biblatex` package entirely uses TeX macros for formatting. BibTeX styles, by contrast, are programmed in a postfix stack language. So, for customizing styles with the `biblatex` package, you don't need to learn another language. It supports subdivided bibliographies. You can generate multiple bibliographies within one document. You can have separate lists of bibliographic shorthands. The `biblatex` package can still use BibTeX as backend, but works with a new and capable backend called `biber`.

It is highly recommended that you use the `biber` backend with the `biblatex` package for the following reasons:

- It is capable of processing UTF-8 input, including accented characters and Unicode symbols. BibTeX, by contrast, requires 7-bit ASCII text.
- You can customize the sorting, such as by deciding whether or not to take capitalization into account and by following sorting guidelines of various languages. It can automatically use your operating system locale.
- Names can be used more flexibly; for example, to cite names with prefixes correctly.
- Multiple bibliographies in the same document, for example, in the same chapter, can be produced in one run, while BibTeX needs one run for each chapter.
- The `biber` backend can even sort each bibliography in the document independently and differently.
- It doesn't have memory limitations, which can be a problem with BibTeX.

How to do it...

We will start with a small example, which is the same as the one in *LaTeX Beginner's Guide*, so you can compare the new `biblatex` package approach to the classic BibTeX way, if desired.

Follow the following steps:

1. With any text editor or the LaTeX editor, open a new document and add these bibliography entries:

```
@book{DK86,
  author = "D.E. Knuth",
  title = "The {\TeX}book",
  publisher = "Addison Wesley",
  year = 1986
}
@article{DK89,
  author = "D.E. Knuth",
  title = "Typesetting Concrete Mathematics",
  journal = "TUGboat",
  volume = 10,
  number = 1,
  pages = "31--36",
  month = apr,
  year = 1989
}
@book{Lamport,
  author = "Leslie Lamport",
  title = "\LaTeX: A Document Preparation System",
```

```
  publisher = "Addison Wesley",
  year = 1986
}
```

2. Save that document and choose any filename. Here, we give the name `texbooks.bib`.
3. Now, we start the LaTeX document. Begin with a document class:

   ```
   \documentclass{scrartcl}
   ```

4. Load the `biblatex` package:

   ```
   \usepackage{biblatex}
   ```

5. Add the bibliography file with the full filename, including the extension:

   ```
   \addbibresource{texbooks.bib}
   ```

6. Start your LaTeX document body. For citing references, use the `\autocite` command:

   ```
   \begin{document}
   \section*{Recommended texts}
   To study \TeX\ in depth, see \autocite{DK86}. For
   writing math texts, see \autocite{DK89}. The
   basic reference for \LaTeX\ is \autocite{Lamport}.
   ```

7. Insert the following command where you would like to print the bibliography:

   ```
   \printbibliography
   ```

8. Finish your document:

   ```
   \end{document}
   ```

9. Compile the document and take a look at it:

Recommended texts

To study TEX in depth, see [**DK86**]. For writing math texts, see [**DK89**]. The basic reference for LATEX is [**Lamport**].

10. There's still no bibliography. For the cited references, we see just the keys of the bibliography entries, **DK89** and **Lamport**.
11. Run the `biber` backend on the document. Either click on the corresponding editor button, if your editor provides one, or run it at the Command Prompt. Use the document name as argument, but without the `.tex` extension:

    ```
    biber filename
    ```

12. Compile the document again, and take another look at it:

Recommended texts

To study TEX in depth, see [1]. For writing math texts, see [2]. The basic reference for LATEX is [3].

References

[1] D.E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[2] D.E. Knuth. "Typesetting Concrete Mathematics". In: *TUGboat* 10.1 (Apr. 1989), pp. 31–36.

[3] Leslie Lamport. *LATEX: A Document Preparation System*. Addison Wesley, 1986.

How it works...

The `biblatex` package and the `biber` backend support the classic bibliography file format known from BibTeX. It's described in *LaTeX Beginner's Guide*, but also in the BibTeX reference manual and in the `biblatex` package manual. Refer to the latter for complete information.

First, we need to load the package. There are a lot of options to customize, which you can specify when loading the packages via use package. Some of the most useful options are as follows:

- `backend`: This can be the `biber` backend (the default option), `bibtex`, `bibtex8` (the 8-bit version), or `bibtexu` (the unsupported unicode version)
- `style`: This is the name of the style for the bibliography and citations
- `bibstyle`: This is the name of the bibliography style
- `citestyle`: This is the name of the citation style
- `natbib`: This loads commands for citation commands of `natbib`, which is a popular package for author-year citations with BibTeX

You can call the `biblatex` package in the preceding example this way:

```
\usepackage[
  backend = biber,
  style   = authoryear,
  natbib  = true
]{biblatex}
```

You will now get author-year notations, both in citations and in the bibliography, as shown here:

Recommended texts

To study TEX in depth, see (Knuth, 1986). For writing math texts, see (Knuth, 1989). The basic reference for LATEX is (Lamport, 1986).

References

Knuth, D.E. (1986). *The TEXbook*. Addison Wesley.
— (1989). "Typesetting Concrete Mathematics". In: *TUGboat* 10.1, pp. 31–36.
Lamport, Leslie (1986). *LATEX: A Document Preparation System*. Addison Wesley.

The command `\addbibresource` loads your bibliography. It provides several options, given in the `key=value` notation. The most noticeable options are the following:

- `label`: This is the name that you can use instead of the full resource name in a reference section in cases of multiple bibliographies
- `location`: This can be local for a local file (default), or remote for an Internet address, which can be a website or a FTP server, for example
- `datatype`: This is the format of the resource, which can be any of the following:
  - `bibtex`: This is the classic BibTeX format (default)
  - `ris`: This is the Research Information Systems format
  - `zoterordfxml`: This is the Zotero RDF/XML format
  - `endnotexml`: This is the EndNote XML format

You can load several files, even directly from the Internet; for example:

```
\addbibresource{maths.bib}
\addbibresource{history.bib}
\addbibresource[location=remote,
  label=tex]{http://latex-community.org/texbooks.bib}
```

The command `\printbibliography` then prints the sorted and styled bibliography. It understands some optional parameters for customizing it, which are described in the manual.



The `\addbibresource` command replaces the older BibTeX command `\bibliography`. The latter takes just the file name and is not as flexible.

There's more...

While this recipe gives you a quick start, browsing the comprehensive manual is recommended for customization. You can open it, as usual, by typing the `texdoc biblatex` command into the command line, or by visiting `http://texdoc.net/pkg/biblatex`. In the same way, you can access the `biber` manual.

The `biblatex` manual lists a lot of other predefined styles, such as variations of numeric, alphabetic, and author-year styles.

On the CTAN website (`http://ctan.org/topic/biblatex`), there are a lot of additional `biblatex` styles to download. Check whether or not one already matches your requirements, otherwise, the manual can guide you in making the final adjustment. Luckily, you can use LaTeX for it instead of the BibTeX postfix language.



Both the `biblatex` package and the `biber` backend were heavily developed in sync. Ensure that your versions are compatible with each other. The `biber` manual provides a compatibility matrix in its introduction section.

Adding a glossary

If words in your document require some explanation, you can add a glossary. This is an alphabetical list of words or phrases with their explanations. A possible improvement would be having backreferences to the locations in the text where those words are used.

How to do it...

We will work with the `glossaries` package by following these steps:

1. Start with any document class. For our example, we will use the `scrartcl` class, because we don't start with a paragraph indentation with the `parskip` option. However, you can use the `article` class without options as well:

   ```
   \documentclass[parskip=half]{scrartcl}
   ```

2. Load the `glossaries` package and choose the style called `long3col`:

   ```
   \usepackage[style=long3col]{glossaries}
   ```

3. Use this command to tell the package to create a glossary:

   ```
   \makenoidxglossaries
   ```

4. Create the first glossary entry for the word TeX. Using a `key=value` interface, state the name, a word indicating the sort order because the name is actually a macro. Finally, write a description. All of these should be in curly braces:

```
\newglossaryentry{tex}{
  name = {\TeX},
  sort = {TEX},
  description = {Sophisticated digital typesetting system,
                 famous for high typographic quality of
                 mathematical formulae}
}
```

5. Repeat this for each glossary entry. We will do this for LaTeX and TikZ:

```
\newglossaryentry{latex}{
  name ={\LaTeX},
  sort = {LATEX},
  description = {Document markup language based on
                 \gls{tex}, widely used in academia}
}
\newglossaryentry{tikz}{
  name = {Ti\emph{k}Z},
  sort = {TikZ},
  description = {Extremely capable graphics
                 language for drawing with \gls{tex}}
}
```

6. Open the document:

```
\begin{document}
```

7. Write some text. When you use the words which are part of the glossary, use the command `\gls{label}`:

```
\gls{tikz} works with plain \gls{tex}. However,
it is mostly used with \gls{latex}.
```

8. Use the following command to print the glossary:

```
  \printnoidxglossary
```

9. End the document:

```
\end{document}
```

10. Compile the document twice and take a look at it:

TikZ works with plain TeX. However, it is mostly used with LaTeX.

Glossary

LaTeX	Document markup language based on TeX, widely used in academia	1
TeX	Sophisticated digital typesetting system, famous for high typographic quality of mathematical formulae	1
TikZ	Extremely capable graphics language for drawing with TeX	1

How it works...

The basic procedure for producing a glossary is as follows:

- Define your terms or abbreviations. This can also be done in a separate file which you can load via the `\input` command.
- Reference those entries by using the `\gls` command, like you would do with the `\ref` command for a label.
- Finally, display the glossary list.

The `glossaries` package is very capable. It even provides a lot of predefined layouts, called styles. We chose `long3col` for a `longtable` layout with three columns. There are other `longtable` and `supertabular` styles and several list layouts similar to the standard LaTeX description list.



There's a package with a similar name called `glossary`. This is an older version by the same author, which is now obsolete; so, `glossaries` should be used instead.

All the features of the `glossaries` package are explained in the user guide, which you can access by typing the `texdoc glossaries` command into the command line, or by opening it online at `http://texdoc.net/pkg/glossaries`. If you feel a bit overwhelmed by the amount of reference information, you can read the beginner's guide instead, for a quick start. It can be opened by typing the `texdoc glossariesbegin` command into the command line, or by visiting `http://texdoc.net/pkg/glossariesbegin`.

Creating a list of acronyms

For documents with a lot of acronyms or abbreviations, it is common to have a table showing their short forms and full forms. This allows compact writing and adds convenience for the reader.

This is different from a glossary as we don't list explanations, just the full forms.

How to do it...

We will again use the `glossaries` package. Since the concept of a glossary and a list of acronyms are closely related, it provides an acronym mode too, so we will now use the list that way by performing the following steps:

1. Begin with a document class. It doesn't matter which one. Here, we will take the same class that we took in the previous recipe:

   ```
   \documentclass[parskip=half]{scrartcl}
   ```

2. Load the `glossaries` package and choose the style called `long3col` as we did in the previous recipe. For acronym support, add the `acronym` option:

   ```
   \usepackage[acronym,style=long3col]{glossaries}
   ```

3. Choose an acronym style. We take `long-sc-short` here, where `sc` stands for small caps in the short form:

   ```
   \usepackage[acronym,style=long3col]{glossaries}
   ```

4. Define a new acronym using the command `\newacronym` and three arguments: a label, the short form, and the long form. Since we use lowercase anyway, it's recommended that you write the short form in lowercase:

   ```
   \newacronym{ctan}{ctan}{Comprehensive
     \TeX\ Archive Network}
   ```

5. Define other acronyms as needed in the same way:

   ```
   \newacronym{tug}{tug}{\TeX\ Users Group}
   \newacronym{dante}{dante}{Deutschsprachige
     Anwendervereinigung \TeX}
   ```

6. Use this command to tell the package to create defined glossaries; this also applies to lists of acronyms:

   ```
   \makenoidxglossaries
   ```

7. Start the document:

   ```
   \begin{document}
   ```

8. Write some text. As with the glossary in the previous recipe, for acronyms, use the command `\gls{label}`:

```
The \gls{ctan} has been founded by members of the
\gls{tug} and of the German speaking group ``\gls{dante}''.

The \gls{ctan} project is actually independent of
\gls{tug} and \gls{dante}, but \gls{dante} is still
the main supporter.
```

9. Use this command to print the list of acronyms:

```
\printnoidxglossary[type=\acronymtype]
```

10. End the document:

```
\end{document}
```

11. Compile the document twice and take a look at it:

The Comprehensive TEX Archive Network (CTAN) has been founded by members of the TEX Users Group (TUG) and of the German speaking group "Deutschsprachige Anwendervereinigung TEX (DANTE)".

The CTAN project is actually independent of TUG and DANTE, but DANTE is still the main supporter.

Acronyms

CTAN	Comprehensive TEX Archive Network	1
DANTE	Deutschsprachige Anwendervereinigung TEX	1
TUG	TEX Users Group	1

How it works...

The preceding commands works exactly like the previous recipe about glossaries. The only substantial addition is the following command:

```
\newacronym{label}{short version}{long version}
```

This command defines each acronym.

Producing an index

Longer documents, such as technical books, often contain an index. This is usually placed at the end of the document. It contains a list of words and the page numbers at which a reader can find them in the document. Such a list may contain key words, topics, or person's names.

How to do it...

Indexing may be done at the very end. Reading through the document, a command `\index{keyword}` should be placed for each place relevant to that key word. A good place can be right before the occurrence of that key word in the text. Don't leave a space between the `\index{...}` command and the indexed word.

We will use the `makeidx` package. We will now insert the indexing commands while writing by following these steps:

1. Start with a document class. We will use the same class that we used in the previous recipes:

   ```
   \documentclass[parskip=half]{scrartcl}
   ```

2. Load the `makeidx` package:

   ```
   \usepackage{makeidx}
   ```

3. Use the this command to generate the index:

   ```
   \makeindex
   ```

4. Begin the document:

   ```
   \begin{document}
   ```

5. Write the text, and insert the `\index` commands right before key words:

   ```
   While \index{glossary}glossary entries are simply
   printed in the text, an \index{acronyms}acronym
   is firstly fully printed with the short version
   in parentheses, later only in the short version.
   ```

6. Use this command to print the index:

   ```
   \printindex
   ```

7. End the document:

   ```
   \end{document}
   ```

8. Compile the document for first time.

9. Run the `makeindex` command in your editor or execute the command `makeindex filename` at the command line.
10. Compile again with LaTeX, and take a look at the document:

While glossary entries are simply printed in the text, an acronym is firstly fully printed with the short version in parentheses, later only in the short version.

Index

acronyms, 1

glossary, 1

How it works...

All your `\index` commands generate entries with the key word and page number in an auxiliary file. This is processed by the external program `makeindex`. In the second run, its output is read in and printed.



The `latexmk` tools can automate this process. So, you just need a single call of `latexmk`, which calls LaTeX for you, and even the `biber` backend or `bibtex`, and `makeindex`, if required. These command runs often enough to resolve cross references. You can read about this by typing the `texdoc latexmk` command in the command line, or by visiting `http://texdoc.net/pkg/latexmk`.

There's more...

Indexing is rather a final task. If you would like to add index entries on the fly, you can create macros to help with it. A popular application, for example, is defining a key word macro that prints a word in a certain style and adds an index entry at the same time.

Index entries can have subentries. They are added behind the main entry separated by an exclamation mark, as follows:

```
\index{main topic:sub topic}
```

We can put the tasks of formatting and creating subentries into a sample macro for packages, which does the following:

- It creates a subentry for the package topic
- It creates a subentry for a given topic
- It prints it with some formatting

This can be illustrated by the following commands:

```
\newcommand{\package}[2]{\index{packages!\texttt{#2}}%
  \index{#1!package \texttt{#2}}\texttt{#2}}
```

Let's now extend our text, so we get more index entries to play with even though it may look a bit overdone:

```
While \index{glossary}glossary \index{glossary!entry}entries are
simply printed in the text, an \index{acronym}acronym is firstly
fully printed with the \index{acronym!short version}short version
in \index{parentheses}parentheses, later only in the short
version.

A glossary can be done using one of the packages
\package{glossary}{glossaries} or \package{glossary}{nomencl}.
For a list of acronyms, suitable are \package{acronyms}{acronym},
\package{acronyms}{acro} and also \package{acronyms}{glossaries}.
```

Compile twice; the output now changes to the following:

While glossary entries are simply printed in the text, an acronym is firstly fully printed with the short version in parentheses, later only in the short version.

A glossary can be done using one of the packages `glossaries` or `nomencl`. For a list of acronyms, suitable are `acronym`, `acro` and also `glossaries`.

Index

acronym, 1
    package `acronym`, 1
    package `acro`, 1
    package `glossaries`, 1
    short version, 1

glossary, 1
    entry, 1
    package `glossaries`, 1
    package `nomencl`, 1

package
    `acronym`, 1
    `acro`, 1
    `glossaries`, 1
    `nomencl`, 1
parentheses, 1

You can see subentries and typewriter formatting for package names.

8
Getting the Most out of the PDF

In this chapter, we will explore the features of PDF. We will cover recipes for the following tasks:

- Adding hyperlinks
- Adding metadata
- Adding copyright information
- Inserting comments
- Producing fillable forms
- Optimizing the output for e-book readers
- Removing the white margins
- Combining PDF files
- Creating an animation

Introduction

Originally, LaTeX produced output in the **Device independent file** (**DVI**) format. This is still supported, however, today there exist newer page description languages, such as **PostScript** (**PS**) and the most popular **Portable Document Format** (**PDF**).

There are converters from DVI to PS, from PS to PDF, and also directly from DVI to PDF. A modern TeX compiler called **pdfTeX** can directly generate PDF output. This, combined with the LaTeX format is called **pdfLaTeX**, and it is pretty much the standard today.

Adding hyperlinks

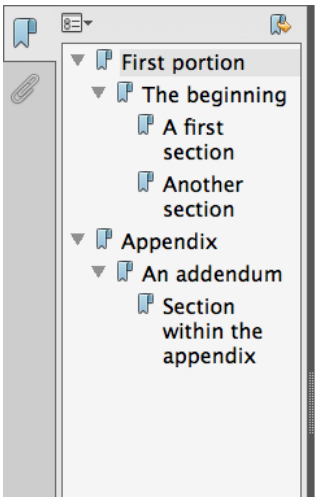
At first, LaTeX was used to produce high-quality prints on paper. Since the output was device independent from the beginning, the results can be viewed on screens, tablets, and smartphones as well.

With electronic publishing, we get access to very handy features. Most importantly, hyperlinks, which allow us to easily navigate within a document. In this recipe, we will deal with providing hyperlinks in the PDF.

How to do it...

We will use the `hyperref` package for providing hyperlinks in the PDF. It provides a user-friendly interface to access PDF features, such as hyperlinks. You can test it with our examples from *Chapter 1*, *The Variety of Document Types*, by performing the following steps:

1. Open the `book` example from the first chapter in your editor.
2. At the end of the preamble, add this line for loading the `hyperref` package:
   ```
   \usepackage{hyperref}
   ```
3. Compile at least twice so that LaTeX can process the data in the `.aux` file written by the `hyperref` package.
4. Take a PDF viewer, such as the Adobe Reader, and take a look at the table of contents:



Contents

I. First portion 5
1. The beginning 7
1.1. A first section 7
1.2. Another section 7
II. Appendix 11
A. An addendum 13
A.1. Section within the appendix 13



You can spot hyperlinks by the red borders around the entries. Each contents entry is a hyperlink now. Click on an entry to move to the corresponding chapter or section in the document body.

5. On the left, you can see bookmarks for additional navigation.

The `hyperref` package does a lot more for us. Let's sum up the most important changes:

- All entries in the table of contents, list of figures and tables, and similarly generated hyperlinks, point to the corresponding place of the object in the body text.
- Footnote markers point to the footnotes. That's the same for endnotes, taking away the pain of scrolling down to the end.
- Citations are hyperlinked to their entry in the bibliography.
- References done using `ref` keyword will become hyperlinks pointing to the location where the label was set. This applies, for example, to figure and table captions, headings, and equations.

How it works...

Though enabling was very easy for us, the `hyperref` package does a lot internally. It redefines many commands of classes and packages for adding hyperlink functionality.



As a rule of thumb, load the `hyperref` package after all other packages to let it override their features. There are exceptions, such as the `cleveref`, `amsrefs`, and packages built on top of the `hyperref` package, such as `bookmark`, `hypcap`, and `hypernat`. You can find a complete list of dependencies at `http://texblog.net/hyperref`.

Like most packages, the `hyperref` package can be controlled using options such as `\usepackage[linkcolor = {blue}]{hyperref}`. It provides a `key=value` interface. However, its specialty is that you can set such options after loading the package, by using the following command:

```
\hypersetup{key1 = value1,
            Key2 = value2,
            ...
}
```

In both cases, some keys can be set without a value, which will then default to `true`.

The `\hypersetup` command is preferable for the following reasons:

- LaTeX expands values in the package options early, which can be undesirable.
- LaTeX strips the spaces from options.
- Some options, such as metadata, are better set after loading the package.
- While curly braces can protect the values in options, you may omit them in the `\hypersetup` command.
- If a document class already loads the `hyperref` package, like the `beamer` package does, or another package, you should not load it again by using the `\usepackage{hyperref}` command. However, you can still use the `\hypersetup` command.

In the next recipe, *Adding metadata*, we will make a comparison between `\hypersetup` and using options in `\usepackage`. Besides this, we will systematically use the `\hypersetup` command. The following pages will be full of examples.

There's more...

In addition to the automatic behavior, the `hyperref` package provides the user with commands for explicitly setting anchors and links. We can customize the link appearance. Over time, the `hyperref` package has become a fully fledged PDF interface. We will also explore some of its PDF specific features.

Inserting custom hyperlinks

You can create your own anchors, give them a name, and link to those anchors. Also, you can link them to Internet addresses.

Linking to a place within the document

Take a look at this short example:

```
\documentclass{article}
\usepackage{hyperref}
\begin{document}
See \hyperlink{mytarget}{next page}.
\newpage
\hypertarget{mytarget}{Here} starts a new page.
\end{document}
```

It works this way:

- We chose the internal link name `mytarget`, which is added to a printed piece of text
- We linked this anchor to the printing text with the hyperlink

As you can see, we link to the target even before it has been set. That's why we need two compiler runs as LaTeX doesn't know the target in the first run before it is defined.

Linking to labeled objects

A single command is sufficient where we have already defined a label, as we you see here:

```
\begin{equation}
  \label{eq:einstein}
  E = mc^2
\end{equation}
```

Now, we can simply write the following:

```
Refer to the \hyperref[eq:einstein]{mass-energy equivalence}.
```

We got a hyperlink with text, instead of a number as we would have with the `\ref` command.



We used square brackets for the `label` argument.

A common way to categorize labels is by using prefixes, such as `eq:` for equations and `fig:` for figures. It helps with keeping the code organized, but it is not necessary.

Linking to the Internet

Let's take a quick look at some further commands. We will use samples of real-world addresses instead of placing holders, so that you can for immediately understand the usage:

- The `\href{http://latex-community.org}{LaTeX Community Forum}` command prints text with a hyperlink to an address
- The `\url{http://texample.net}` command prints the formatted address with hyperlink
- The `\nolinkurl{http://texdoc.net}` command prints the formatted address without link

In the preceding explanation, we used the `\hyperref` command for pointing to a label, using square brackets. There's a more complex syntax with curly braces for Internet addresses:

The `\hyperref{address}{category}{name}{text}` command prints the text and links it to the `address#category.name` parameter.

Let's see a sample of this command:

```
\newcommand{\baseaddress}{http://latex-community.org/}
\newcommand{\articleaddress}{\baseaddress
  know-how/latexs-friends/61/486/}
\hyperref{\articleaddress}{running}{lualatex}
  {Running MetaPost within LuaLaTeX}
```

The generated hyperlink points to `http://latex-community.org/know-how/latexs-friends/61/486/#running.lualatex`.



Generally, to avoid confusing the arguments, here is a mnemonic:

- The first argument is the name or label of a target, which is invisible, and the last argument, if needed, is the text which is displayed

Changing the color and shape of the hyperlinks

The default highlighting by red borders can seem a bit obtrusive. You can simply remove them by adding the `hidelinks` option:

```
\hypersetup{hidelinks}
```

Users may know what a reference is, thus is clickable. Furthermore, we can see the mouse pointer changing when we hover over a link.

However, you can choose to color the text of hyperlinks instead of applying the default frames:

```
\usepackage{xcolor}
\hypersetup{
  colorlinks,
  linkcolor = {red!75!black},
  citecolor = {green!40!black},
  urlcolor  = {blue!40!black}
}
```

Here, we loaded the `xcolor` package to use its color mixing syntax. We get a color for document internal links with 75 percent red plus 25 percent black. Then, we set color for citations, and then for external links to the Internet.

Blackening the hyperlinks, and so getting darker colors, may be less obtrusive and look better in print. Colored links are actually printed, in contrast to the default borders, so you may still decide to use the `hidelinks` package for printing. Hyperlinks on classic paper don't need emphasizing.

To get a consistent link color for all kinds of links, you can set the `allcolors` parameter for coloring the hyperlinks instead.

Getting backreferences in the bibliography

To get a backlink for each bibliography item in order to know where it has been cited, you can set the `backref` option:

```
\hypersetup{backref}
```

The section number will be printed and linked at the end of each bibliography item. Valid optional values for the `backref` key are `section`, `slide`, `page`, `none`, and `false`.

If you would like to have backlink page numbers at the end of bibliography items, use this:

```
\hypersetup{pagebackref}
```

Hyperlinking index entries

Since an index is for looking things up, hyperlinks to the corresponding places in the text are quite natural. It is not enabled by default. However, you can switch it on by using the following:

```
\hypersetup{hyperindex}
```

Adding metadata

The PDF standard contains a field for hidden descriptive data, such as the name of the author and the title of a document. This is useful for archiving purposes. Internet search engines also inspect metadata information. In this recipe, we will edit it.

How to do it...

We will use the interface of the `hyperref` package. Let's again use our examples from *Chapter 1*, *The Variety of Document Types*. Let's take a look at the following steps:

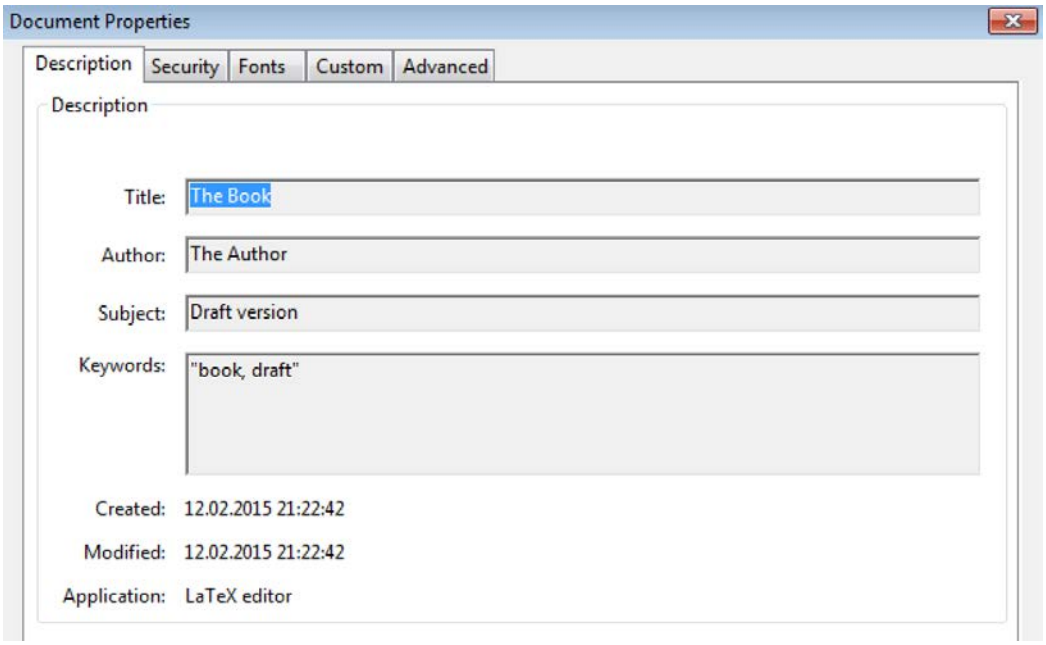
1. Open the `book` example from *Chapter 1*, *The Variety of Document Types*, in your editor.
2. At the end of the preamble, load the `hyperref` package:

   ```
   \usepackage{hyperref}
   ```

3. Set up the metadata information. Here, we use some dummy text:

```
\hypersetup{pdfauthor   = The Author,
            pdftitle    = The Book,
            pdfsubject  = Draft version,
            pdfkeywords = {book, draft},
            pdfproducer = TeX version,
            pdfcreator  = LaTeX editor}
```

4. Compile your document.
5. Take a PDF viewer, such as the Adobe Reader, and inspect the document properties. You can find this meta information here. It will look like this:





In a metadata field, the Adobe Reader adds quotes whether it contains separators such as commas or quotes.

How it works...

Some metadata is usually set automatically, such as the pdfTeX compiler version, which produces a PDF. You can overwrite it and set your own data using the `\hypersetup` interface.

As commas separate options, we need to protect values by enclosing them in curly braces if they contain commas, such as the keywords in our example.

There's more...

In the first recipe, we explained why the `\hypersetup` command is preferable to options to the `\usepackage` package. Let's take a look at how it looks in the latter case, especially for preserving the spaces in values:

```
\usepackage[pdfauthor   = {The\ Author},
            pdftitle    = {The\ Book},
            pdfsubject  = {Draft\ version},
            pdfkeywords = {{book, draft}},
            pdfproducer = {TeX\ version},
            pdfcreator  = {LaTeX\ editor}]{hyperref}
```

We should group by braces to be on the safe side with option parsing. The space with a preceding backslash is a forced space, otherwise the space would get lost. It looks a bit more complicated, especially if you set a lot of other options.

There's an alternative interface for setting metadata that lets you even set your own keys. This is the `pdfinfo` option, which works this way:

```
\hypersetup{pdfinfo = {
              Author   = The Author,
              Title    = The Book,
              Subject  = Draft version,
              Keywords = {book, draft},
              Producer = TeX version,
              Creator  = LaTeX editor,
              Version  = 2.0,
              Comment  = Contains dummy text}}
```

We set the known data and add our own keys.

For completeness, we can even declare metadata without the `hyperref` package by using a pdfTeX command if this compiler is used:

```
\pdfinfo{
  /Author (The Author)
  /Title (The Book)
  /Subject (Draft version)
  /Keywords (book, draft)
  /Producer (pdfTeX 1.40.0)
  /Creator (LaTeX editor)}
```

This is obviously an unusual syntax, which calls for using the `hyperref` package.

Adding copyright information

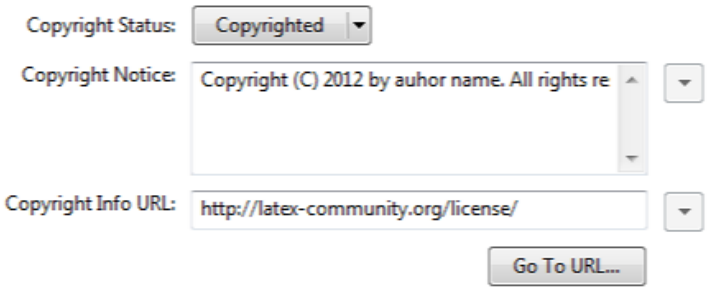
In the previous recipe, we added author, producer, and creator information. We can even add a field for our own copyright information. However, there's a specific place for copyright information. Even after adding metadata like we did earlier, the Adobe Acrobat and other PDF viewers may still display "Copyright Status: Unknown." and an empty copyright notice. We can change this.

How to do it...

The `hyperxmp` package can embed the required information. Follow these steps to add copyright information:

1. Open the `book` example from the first chapter in your editor.
2. At the end of the preamble, load the `hyperref` package:
   ```
   \usepackage{hyperref}
   ```
3. In the next line, load the `hyperxmp` package:
   ```
   \usepackage{hyperxmp}
   ```
4. Set up the copyright information:
   ```
   \hypersetup{
     pdfcopyright = {Copyright (C) 2012 by author name.
       All rights reserved.},
     pdflicenseurl = {http://latex-community.org/license/}}
   ```

5. Compile your document.
6. Take a PDF viewer and inspect the document properties. For example, with the PDF-XChange viewer, click on **File** | **Document Properties**, and then click on the **Additional Metadata** button. You will see the following:



How it works...

Adobe Systems, Inc. supports **eXtensible Metadata Platform** (**XMP**) for embedding metadata. This method takes **eXtensible Markup Language** (**XML**) formatted attributes for embedding within the document.

The `hyperxmp` package processes the `hyperref` package keys and adds further keys, such as those we used earlier in the chapters. So, we don't even need to deal with XML.

Refer to the `hyperxmp` manual for a complete description of all options, including contact data. As usual, you can open it by typing the `texdoc hyperxmp` command in Command Prompt, or by visiting `http://texdoc.net/pkg/hyperxmp`.

There's more...

The `xmpincl` package is more flexible than the `hyperxmp` package, but it doesn't have such a handy interface. You can create a separate XML file, such as `metadata.xmp`, which can be included in the PDF:

```
\usepackage{xmpincl}
\includexmp{metadata}
```

You can also refer to the package manual for complete details and a sample `.xmp` file, as XML markup is beyond the scope of this book and the `hyperxmp` package already solves the task of our recipe.

Inserting comments

If you work collaboratively, you may want to add notes or comments to a document to share additional information with your coauthor which will be removed from the final version. The PDF standard supports comments, and so does LaTeX.

How to do it...

We will insert some comments into our small document example from *Chapter 1*, *The Variety of Document Types*. For brevity, we will directly look at the essential commands. As with all recipes, the full source code can be downloaded from `http://latex-cookbook.net`. Follow these essential steps for inserting comments:

1. Load the `xcolor` package in the preamble to get color support:

   ```
   \usepackage[svgnames]{xcolor}
   ```

2. Load the `pdfcomment` package in the preamble:

   ```
   \usepackage{pdfcomment}
   ```

   You could define some default setting if you wish to , such as the following. Add them when loading:

   ```
   \usepackage[author={Your name},icon=Note,
     color=Yellow,open=true]{pdfcomment}
   ```

3. To insert a simple comment with a marker symbol at a certain place in the document, call the `\pdfcomment` command right at that position:

   ```
   \pdfcomment{Simple documents don't have chapters.}
   ```

4. You can let the comment marker appear in the margin instead:

   ```
   \begin{equation}
     \pdfmargincomment{The equation environment produces
       a centered equation with whitespace before
       and after it.}
     ...
   \end{equation}
   ```

5. To mark visible content while displaying a comment, select it by grouping it within curly braces and use the `\pdfmarkupcomment` command on it. Here, we highlight and comment the word `sections`:

```
The text will be divided into
\pdfmarkupcomment{sections}{You could additionally
  use subsections.}.
```

6. Markup comments can be tooltips that become visible when we move the mouse pointer over them. Let's take a look at the following command:

```
\pdftooltip{formulas}{Formulas can be inline or
  displayed in their own paragraph}
```

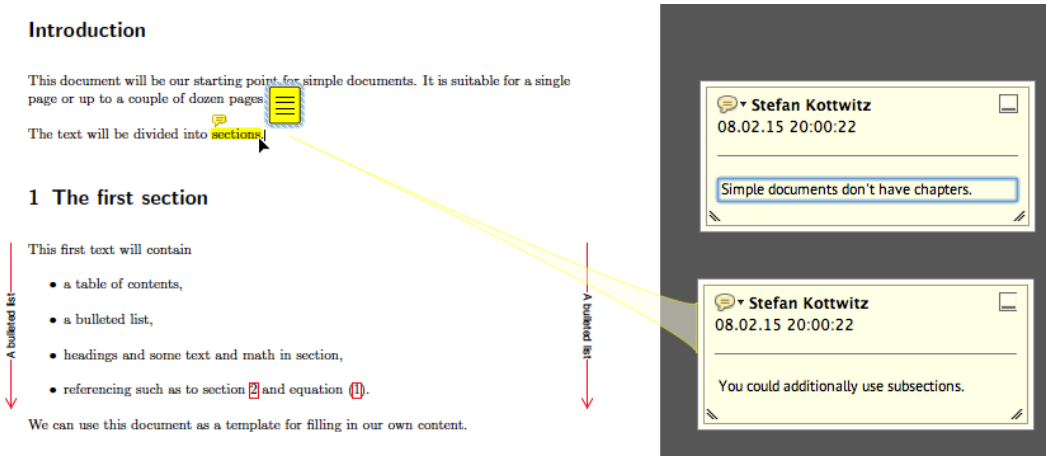
7. You can embrace whole environments with a side line comment:

```
\begin{pdfsidelinecomment}[color=Red]{A bulleted list}
  \begin{itemize}
    \item ...
  \end{itemize}
\end{pdfsidelinecomment}
```

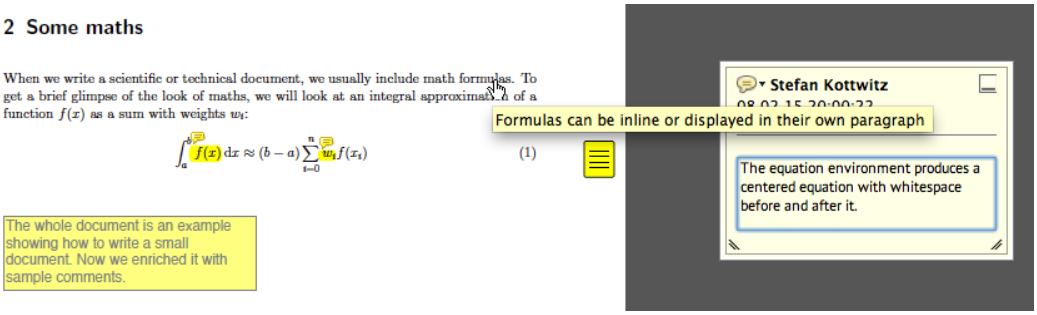
8. Furthermore, you can place free text somewhere in the document with custom dimensions, color, and transparency, by using the following code:

```
\pdffreetextcomment[subject={Summary},width=7.5cm,
  height=2.2cm,opacity=0.5,voffset=-3cm]{The whole document
  is an example showing how to write a small document.
  Now we enriched it with sample comments.}
```

9. With all those comments put into our small sample document, compile twice and examine the results. Here's a simple comment, a markup comment, and a side line comment:



Here, you can see a tooltip, a side comment, markup comments in math mode, and a free text comment:



10. To make all comments invisible in the final published version, add the `final` option:

```
\usepackage[...,final]{pdfcomment}
```

How it works...

There are a lot of style options for customizing the appearance of comments. You can see a selection here in this recipe. The full set is described in the `pdfcomment` package's manual. As usual, you can open it by typing the following at the command prompt:

```
texdoc pdfcomment
```

You can apply such options globally, as we did with the `\usepackage` package in the second step, or locally. In that case, use the options within square brackets as we would usually do in LaTeX:

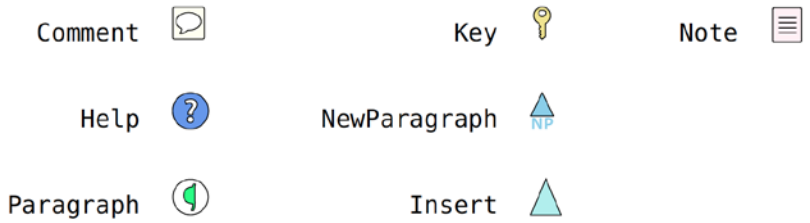
```
\pdfcomment[icon=Insert,color=red,opacity=0.5,
  author=Me]{The comment}
```

Also execute the following command:

```
\pdfmarkupcomment[markup=StrikeOut,color=red]{Text}{The comment}
```

While some options, such as the `width`, `height`, and `color`, are self-explanatory, let's take a final look at the specialties.

The `icon` option can have these values, according to the PDF standard, taken from the manual:



Further icons are supported by the Adobe Reader:



As a `markup` option, you can choose between `Highlight`, `Underline`, `Squiggly`, and `StrikeOut`.

Producing fillable forms

PDF files can be interactive. Users can fill forms before printing the entire document. With LaTeX, you can produce such forms. In this recipe, we will create a form.

How to do it...

Once again, we will use the `hyperref` package. Our goal is to produce small paper sheets for a survey as a fillable PDF form. A yellow background will distinguish it from other papers. The format will be landscape. Let's begin the process:

1. Start a new document with A6 paper in landscape format and with a small interparagraph space instead of paragraph indentation. Let's take a look at the following command:

   ```
   \documentclass[a6paper,landscape,parskip=half]{scrartcl}
   ```

2. Set a small margin to save space on screen:

```
\usepackage[margin=0.4cm]{geometry}
```

3. Set the background color to 30 percent yellow:

```
\usepackage{xcolor}
\pagecolor{yellow!30}
```

4. Choose an `empty` page style, so you won't get page numbers:

```
\pagestyle{empty}
```

5. Load the `hyperref` package:

```
\usepackage{hyperref}
```

6. Start the document and start a `Form` environment:

```
\begin{document}
\begin{Form}
```

7. For positioning, we will use a `tabular` environment. Within that, we use commands for form elements, such as `\TextField`, `\ChoiceMenu`, `\CheckBox`, and `\PushButton`:

```
\begin{tabular}{|lr|}\hline
  \textbf{Dear \TeX\ user, please help in our survey.} &
    \PushButton[width=1cm, onclick =
      {app.alert("You may use a pseudonym for the
      name.")}]{Info}\\[0.5ex] \hline
    & \\
    \TextField[width=5cm]{Name:} &
    \TextField[width=3cm]{Profession:} \\
    & \\
    \ChoiceMenu[radio, radiosymbol=6,
      width=0.5cm]{Software:\quad}{\TeX\ Live, MiK\TeX}
    &\ChoiceMenu[combo, width=3cm]{Editor:}%
    {TeXworks,TeXstudio,TeXmaker,TeXshop,
    WinEdt,Kile,Emacs,vi} \\
    & \\
    Membership in: \hfill\CheckBox[width=0.5cm]{TUG}\hfill
    \CheckBox[width=0.5cm]{DANTE e.V.} &
    \TextField[width=3cm]{Other:}\\ & \\ \hline
  \end{tabular}
```

8. Announce use of the remaining space for notes:

```
\par
\textbf{Notes:}
\par
```

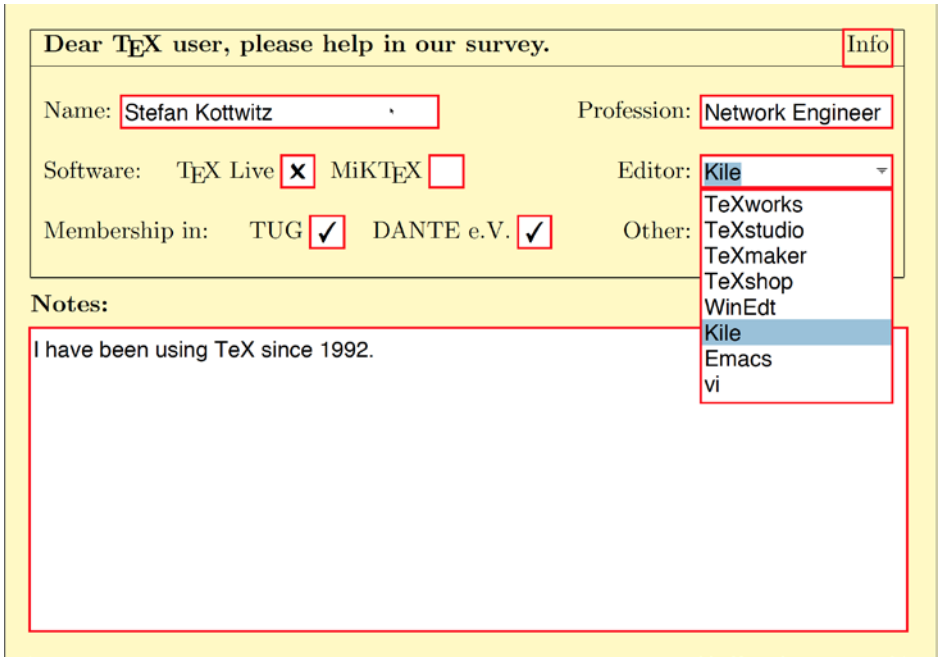
9. Provide a text field where the user will be able to write several lines:

```
\TextField[multiline, width=0.94\paperwidth,
  height=10\baselineskip]{ }
```

10. End the form and the document:

```
\end{Form}
\end{document}
```

11. Compile the document and test it by filling in some values using a capable PDF viewer, such as the Adobe Reader:



12. Save the PDF file, close it, and open it to verify that it stores your data.

How it works...

We used the following three commands for fillable elements:

- `\TextField[options]{label}`: This gives a fillable text field
- `\CheckBox[options]{label}`: This prints a box that can be checked or unchecked
- `\ChoiceMenu[options]{label}{list of choices}`: This produces a drop-down menu where we can choose an element of a list

Some of the possible options for creating a form are as follows:

- `width`, `height`, `borderwidth`: Dimensions of the element
- `charsize`: The font size of the element text
- `color`, `backgroundcolor`, `bordercolor`: Colors of the element
- `maxlen`: The maximum allowed characters in a text field
- `value`: The initial value, such as default text in a field
- `combo`, `radio`, `popdown`: `True` or `false`, type of element
- `radiosymbol`: The Zapf Dingbats symbol for radio button
- `checked`: `True` or `false`; it decides if field is selected by default

Use these options using a `key=value` syntax, like we do in our example. All possible options are described in the `hyperref` package's manual.

We also placed a button in our form by using the following line of code:

```
\PushButton[onclick={app.alert(text)}]{label}
```

This command prints a button with the `label` text on it. If it is clicked, it opens a pop-up window via JavaScript, displaying `text`. You can also add options as we did in step 7 of this recipe.

You can use other JavaScript commands; for example, to let the user enter text in the pop-up window:

```
\PushButton[width=1cm, onclick =
  {app.response("What was your original motivation
  to start with LaTeX?")}]{Question}
```

Even an entire program can be inserted; for example, to use and evaluate the input. You can also bind JavaScript code to other actions, such as `onmouseover` and `onselect`. All of these event handlers are listed in the `hyperref` manual.

For more information on JavaScript supported by Adobe, visit `http://www.adobe.com/devnet/acrobat/javascript.html`.



The `Form` commands should only be used within a `Form` environment. There should be no more than one `Form` environment in the file. However, it can be extensive.

Optimizing the output for e-book readers

A traditional book is a set of pages; LaTeX, too, follows this tradition. However, today, e-book reading mediums such as the Kindle, iPad, tablets, and even smartphones, are popular for browsing the Internet and for reading documents.

In this recipe, we will see how to make a document e-reader-friendly.

How to do it...

Set up a preamble for a document that can be read on an electronic device such as a tablet reader, by performing the following steps:

1. Choose a suitable document class. Choose small headings and a small interparagraph skip instead of paragraph indentation. The latter costs too much space on an already narrow display:

```
\documentclass[fontsize=11pt,headings=small,
  parskip=half]{scrreprt}
```

2. Set a small paper size that matches a common screen ratio and choose a very small margin:

```
\usepackage[papersize={3.6in,4.8in},margin=0.2in]{geometry}
```

3. Choose a font that is easily readable on screen, especially with a low screen resolution; a sans-serif font may be a good choice:

```
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\renewcommand{\familydefault}{\sfdefault}
```

4. Load the `microtype` package for improving justification, which is even more important for smaller screens:

```
\usepackage{microtype}
```

5. Choose the `empty` page style. If the chapter or some of the pages of the chapter have their own style, set those to `empty` as well:

```
\pagestyle{empty}
\renewcommand{\partpagestyle}{empty}
\renewcommand{\chapterpagestyle}{empty}
```

6. Use the `hyperref` package for easier browsing via hyperlinks. Colored links are to be preferred to the framed appearance, as you saw in the first recipe of this chapter:

```
\usepackage{hyperref}
\hypersetup{colorlinks}
```

7. In your document, use relative sizes instead of absolute values, as follows:

```
\includegraphics[width=0.8\textwidth]{text}
```

How it works...

A tablet or smartphone is very different from a book, even though some reading software simulates the traditional feel, such as page flipping. Our approach honors the difference.

We chose a document class that works in one-sided mode. There's no need to insert blank pages to let chapters start on *right-hand* side pages.

With a KOMA-Script class, you can simply choose smaller headings than the default size by setting the class option to `scrreprt`. With other classes, you can change headings and their spacing by using the `titlesec` package.

The default paragraph indentation makes our already-narrow text even harder to justify, so we will remove that. Setting a `parskip` option does this automatically. We have even used the `half parskip` option to reduce the spacing. With a non-KOMA-Script class, load the `parskip` package.

We don't need excessive margins added to the mechanical margins of the reading device; that's why we used the geometry package with very small margins.

Choosing a font is a matter of taste. Serif fonts often look poor with low-resolution displays; that's why sans-serif fonts were often preferred. However, today's screens offer very high resolution, so the decision depends on your taste.

Since reader software displays page numbers, we don't need to reserve space for them. The `empty` page style does this for us.

There's more...

We can be more consequent in designing for e-books. Why turn pages when you can scroll down endlessly? We can forget all the challenges raised by page breaks, such as the concept of floating figures and tables. They can stay where we place them, because no page breaks will get in their way. Boris Veytsman and Michael Ware described an approach in *Ebooks and paper sizes: Output routines made easier* in *TUGboat, Vol. 32, No.3*, which is available at `https://www.tug.org/TUGboat/tb32-3/tb102veytsman-ebooks.pdf`.

The `screenread` package implements this idea; you can find it at `http://www.phys.psu.edu/~collins/screenread/`.

Removing white margins

To reuse the PDF output of your LaTeX document in another document, in an e-book, or on a website, it's usually a good idea to remove the margins, or at least to make them smaller.

Getting ready

For this recipe, you need to have the following things installed:

- The PostScript interpreter software—**Ghostscript**
- The programming language—**Perl**
- The `pdfcrop` script

Fortunately, TeX Live installs internal versions of Ghostscript and Perl by default, and offers the `pdfcrop` script as a package; so, TeX Live users are lucky. If you don't use TeX Live, you may need to install the software. You can find more information at `http://www.ghostscript.com`, `http://www.perl.org` and `http://ctan.org/pkg/pdfcrop`.

How to do it...

The procedure is pretty easy, given that Command Prompt won't be an obstacle for you. We use LaTeX, so this won't be a challenge. Follow these steps for removing the white margins:

1. Open Command Prompt.
   - On Linux or Unix, open a **shell**
   - On Ubuntu, Linux, and Mac OS X, the shell you need to open is called **Terminal**
   - On Windows, open **Command Prompt** or run the program called `cmd`
2. Change into the directory of your generated PDF file.
3. Given that the document name is `filename.pdf`, execute the following command:

   ```
   pdfcrop filename
   ```

4. The `filename-crop.pdf` file has been generated, open it to verify the result.

How it works...

The `pdfcrop` program takes the PDF file and uses Ghostscript to calculate the bounding box for each page. It produces another PDF file in which the margins are removed.

The output is named with `–crop` as a name suffix by default. However, you can give the output file name as parameter in the command. The syntax to be called is as follows:

```
pdfcrop [options] filename[.pdf] [outputname]
```

You don't need to tell the program the `.pdf` file extension for the input file.

Using these options, you can decide to keep some margins:

- Keep a 20 PostScript point margin on each side:

  ```
  pdfcrop --margins 20 input.pdf output.pdf
  ```

- Keep 10, 20, 30, and 40 PostScript points at the left, top, right, and bottom, respectively:

  ```
  pdfcrop --margins '10 20 30 40' input.pdf output.pdf
  ```

A PostScript point is 1/72 inches; in TeX, it's called a **big point** and is written as 1 bp. The classical TeX point is slightly different, where 1 pt means 1/72.27 inches.

You can display options and information by using the following command:

```
pdfcrop –help
```

There's more...

You can remove the margins directly from the source code.

If the class doesn't matter, for example, because you are just generating a graphic, you can use the `standalone` class:

- For a document without margins, use the following:

  ```
  \documentclass{standalone}
  ```

- To get a 10 pt margin, use the following:

  ```
  \documentclass[border=10pt]{standalone}
  ```

- For margins of 10 pt, 20 pt, 30 pt, and 40 pt at the left, right, bottom, and top, respectively, use the following:

  ```
  \documentclass[border={10pt 20pt 30pt 40pt}]{standalone}
  ```

You can freely choose another unit, such as `in` or `cm`.

The `standalone` class provides more handy features and is well documented. You can read the manual via `texdoc standalone`, or online at `http://texdoc.net/pkg/standalone`.

The older `preview` package does a similar job. It can extract environments of a LaTeX document as separate graphic files.

Combining PDF files

Combining the source code of two LaTeX documents can be pretty hard, especially if they are based on different classes. However, combining their PDF output is pretty easy.

How to do it...

We will use the `pdfpages` package.

You can test it with the flyer example from *Chapter 1*, *The Variety of Document Types*, together with the form example from the current chapter. Let's get going:

1. Open a document and choose any class:
   ```
   \documentclass{article}
   ```
2. Load the `pdfpages` package:
   ```
   \usepackage{pdfpages}
   ```
3. Begin the document:
   ```
   \begin{document}
   ```
4. Include the first PDF file by using the command `\includepdf`. It takes the page range as an option. Use a dash `(-)` for the whole page range:
   ```
   \includepdf[pages=-]{flyer}
   ```
5. Include the second PDF file:
   ```
   \includepdf[pages=-]{form}
   ```
6. End the document:
   ```
   \end{document}
   ```
7. Compile once, and look at the newly generated file, which contains both the flyer and the form.

How it works...

The `pdfpages` package is primarily used for including PDF files in a LaTeX document, completely or partially. The `pages=-` option means all pages. We can specify a page range, such as `pages={3-6}`, for more complex sets, such as the `pages={1,3-6,9}` page.

We created a new document as a container and simply included both source PDF files in it.

This way, you can combine very different files, such as an application letter, curriculum vitae, and various scanned certificates, into a single file for an online job application.



If the included PDF would have hyperlinks, they would get lost. Instead of combining in a third document, you could include the simpler document, such as a cover PDF file, in the more complex document.

Creating an animation

To show a developing process or to demonstrate changes, an in-place animation can be more convenient than a series of images.

As an example application, we will draw a recursively-defined fractal curve, the Koch curve. An animation will present the stages of the curve, which become more complex with higher numbers of recursions.

How to do it...

The `animate` package provides a simple way to generate an animation. Try this with the Koch curve, to show growing complexity by performing the following steps:

1. Start with any document class. Here, we choose the `standalone` class, which we already mentioned earlier. Here, the paper tightly fits the animation:

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `animate` package:

   ```
   \usepackage{animate}
   ```

3. Load the TikZ package. Furthermore, load the `lindenmayersystems` library for producing fractals, and the `shadings` library to fill with a shading:

   ```
   \usepackage{tikz}
   \usetikzlibrary{lindenmayersystems,shadings}
   ```

4. We define the fractal with the library's syntax. Don't worry about it, as it's just for having stuff to play with. Let's take a look at the following commands:

   ```
   \pgfdeclarelindenmayersystem{Koch curve}{
     \rule{F -> F-F++F-F}}
   ```

5. Start the document:

```
\begin{document}
```

6. We use an `animateinline` environment. We specify options to show control buttons to automatically start the animation, and to loop. This means that the animation restarts. The mandatory argument in curly braces is the speed; here, two frames per second:

```
\begin{animateinline}[controls,autoplay,loop]{2}
```

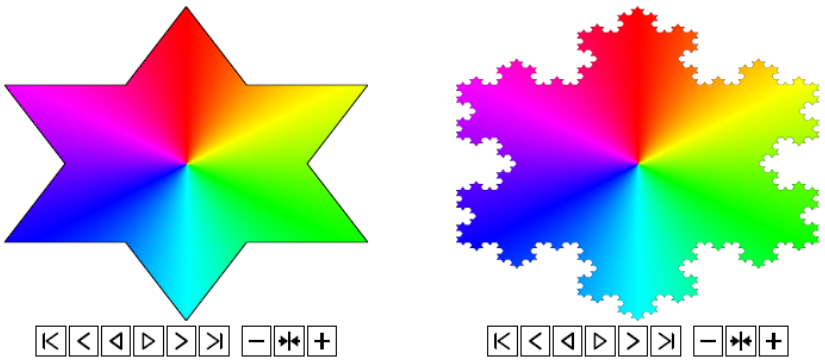
7. We use the `\multiframe` command to produce five frames, which we state as the first argument. The second argument is a variable, `n`, starting at `0` and incrementing by `1` in each step. Within this command, we will print the curve of order `\n`. Just take the TikZ curve code as it is, since we are focusing on the animation:

```
\multiframe{5}{n = 0+1}{
  \begin{tikzpicture}[scale = 80]
    \shadedraw[shading = color wheel]
    [l-system = { Koch curve, step=2pt, angle=60,
    axiom=F++F++F, order=\n }]
    lindenmayer system -- cycle;
  \end{tikzpicture}
}
```

8. End the `animateinline` environment and the document:

```
\end{animateinline}
\end{document}
```

9. Compile the document and open it in a compatible PDF viewer, such as Adobe Reader. Here is the output of steps 2 and 5 of the animation. You can click on the buttons to play backward and forward, and to increase or decrease the speed:



How it works...

We used two main features of the package:

- The `animateinline` environment creates an animation from the environment's content, which means not from external files. Any drawing package can be used, such as TikZ or PSTricks. We used TikZ here.
- The `\multiframe` command lets us build a loop around drawing commands. The syntax is as follows:

```
\multiframe{number of frames}{variable =
  initial value+increment}{drawing code with \variable}
```

The rest is drawn with TikZ. The `\shadedraw` command creates the actual drawing while filling it with shading. Here, we used a shiny shading called `color wheel`. It may show a nice glimpse of the TikZ package, while we focus on animating it. In *Chapter 9, Creating Graphics*, you will see more of TikZ. There's also a link to the manual, which explains the `lindenmayersystems` library.

With the `animate` package, you can also assemble animations from the existing image files. The command for this is as follows:

```
\animategraphics[options]{frame rate}{file basename}{first}{last}
```

In this command, within square brackets, options can be like the preceding options that are shown. If you set the `file basename` option to `frame-`, `first` to `4`, and `last` to `12`, the command will play the sequence from `frame-4.png` to `frame-12.png`, if they exist in a PNG format. The filename extension is used automatically, so, with pdfLaTeX, it will look for PNG, JPG, and PDF files.

For further details, you can take a closer look at the manual, which you can open by typing the `texdoc animate` command in to the Command Prompt, or you can find it at `http://texdoc.net/pkg/animate`.

9
Creating Graphics

Today, LaTeX provides comprehensive graphic capabilities. In this chapter, we will work on some impressive graphics.

Specifically, this chapter covers the following recipes:

- Building smart diagrams
- Constructing a flowchart
- Growing a tree
- Building a bar chart
- Drawing a pie chart
- Drawing a Venn diagram
- Putting thoughts in a mind map
- Generating a timeline

Introduction

A picture is worth a thousand words. By presenting a single image, you can visualize a complex concept so that it becomes much easier to understand. Especially using diagrams, you can distill relevant information, and, for example, show relationships and processing orders or compare quantities.

This chapter mainly contains recipes for several kinds of diagram. We will start with simple diagrams and trees, then we will produce charts from data, finally we will build diagrams about concepts and events over time.

All the recipes in this chapter are based on **pgf**/**TikZ**, which is an enormously capable graphics package. **PGF** stands for **Portable Graphics Format**, which is the backend. **TikZ** is the frontend. The name is an abbreviation of **TikZ ist kein Zeichenprogramm**, which translates to "TikZ is not a drawing program". This recursive acronym, created in the tradition of GNU, should tell potential users what to expect: no WYSIWYG. This means that you cannot see the output during creation, but after compiling. However, with TikZ, you benefit from TeX's strong points: fine quality, macros, reusability, and a large number of existing libraries and packages.

There's another very capable graphics package for LaTeX with a long history and countless features—**PSTricks**. It is based on PostScript, which can be translated into PDF.

In this chapter, we will take the TikZ way, because of its very readable syntax, excellent documentation, and compatibility with all TeX engines (pdfTeX, LaTeX in DVI mode, XeTeX, and LuaTeX) and formats (LaTeX, plain TeX, and ConTeXt). Last but not least, the decision is led by a view of the growing popularity of TikZ in Internet forums over the years, which can also be verified using a Google Trends search.

We won't deal with how to draw rectangles or circles, since this is covered by the manual. Instead, we will start off by using modern packages for various purposes to create complete and useable graphics.

Getting ready

To compile the recipes in this chapter, you will need to have TikZ installed. Note that while the term TikZ is the popular name, the actual package name is `pgf`, which you will be able to see in the package manager of your LaTeX installation. Version 3.0 or later is recommended.

Most recipes use another package on top of TikZ. So, for each package mentioned at the beginning of a recipe, make sure you have it installed. A full TeX installation is recommended anyway.

In order to cover many complete and usable recipes in this chapter about graphics, the basic TikZ commands will not be explained. You can read about them in the reference manual at any time. It's assumed that you already know something about nodes, edges, and styles; take a look at the manual if you need to understand these concepts. Of course, it's explained in detail how the recipes themselves work.

So, for further customization and a deeper understanding, keep the TikZ manual at hand. You can open it simply by typing `texdoc tikz` or `texdoc pgf` in the command line. The same applies to each package used.



If you are reading this book without having a TeX installation with documentation available, visit `http://texdoc.net/pkg/tikz`.

Building smart diagrams

First, let's focus on a quick win: getting a diagram with minimal technical effort. We would just need to fill in our thoughts.

The `smartdiagram` package makes building diagrams of various types very easy. We will start with creating a flowchart about a TeX workflow. Then we will explore further diagram types.

How to do it...

Once you have loaded the `smartdiagram` package, all you need is a simple command. Follow these steps:

1. As always, begin with any document class:

   ```
   \documentclass{article}
   ```

2. Load the `smartdiagram` package:

   ```
   \usepackage{smartdiagram}
   ```

3. Start the document:

   ```
   \begin{document}
   ```

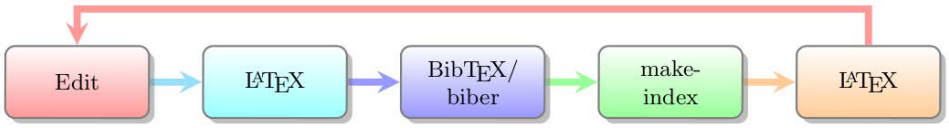
4. Define the diagram. An option in square brackets defines the type, and an argument in curly braces contains a comma-separated list of items:

   ```
   \smartdiagram[flow diagram:horizontal]{Edit,
     \LaTeX, Bib\TeX/ biber, make\-index, \LaTeX}
   ```

5. End the document:

   ```
   \end{document}
   ```

6. Compile, and take a look at the flowchart of a TeX workflow:



How it works...

A simple call of `\smartdiagram` plus arguments produced the image. The syntax of the command is as follows:

```
\smartdiagram[type of diagram]{list of items}
```

We chose a horizontal flow diagram. Use only `flow diagram` to get a vertical layout. There's a specific set of diagram types. Instead of just listing them, let's perform some examples to take a look at the results.

There's more...

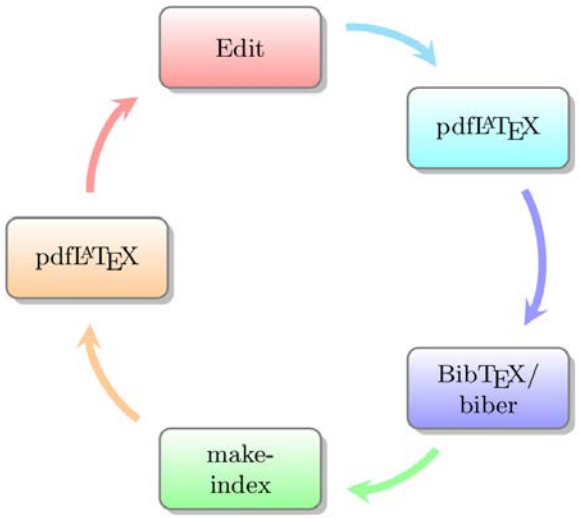
We can play a bit with the diagram type and items to get different diagrams. If there's a root item or a central item, it's always the first in the list.

Circular diagrams

Let's try a circular diagram. It is counterclockwise by default. Add `:clockwise` to the option to get a clockwise order:

```
\smartdiagram[circular diagram:clockwise]{Edit,
  pdf\LaTeX, Bib\TeX/ biber, make\-index, pdf\LaTeX}
```

The following figure shows the circular diagram of a TeX workflow:



Bubble diagrams

To simply present some words in the main context, you can use a bubble diagram. The first item on the list is in a central circle. All the other items are placed in colored circles around it. They are slightly overlapping so as to visualize a close relationship.

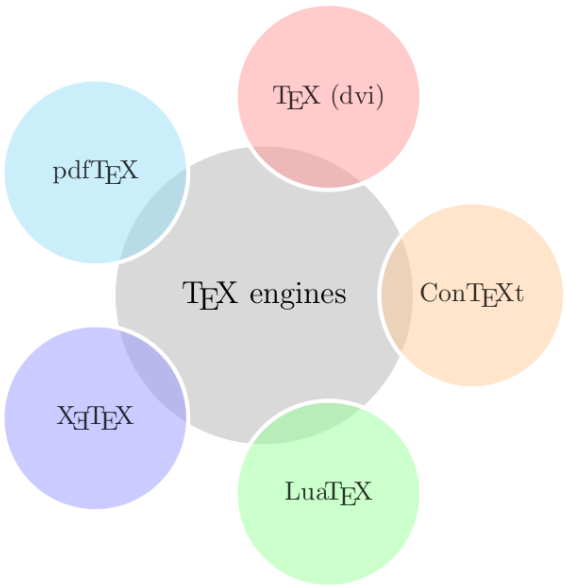
To get macros for more TeX logos, we will load the `dtklogos` package in addition, so add this to the preamble:

```
\usepackage{dtklogos}
```

Alternatively, you can use the `metalogo` package, which gives logo commands such as `\XeTeX`. Combining strings by writing `Lua\TeX` instead would be another idea.

The `smartdiagram` call becomes this:

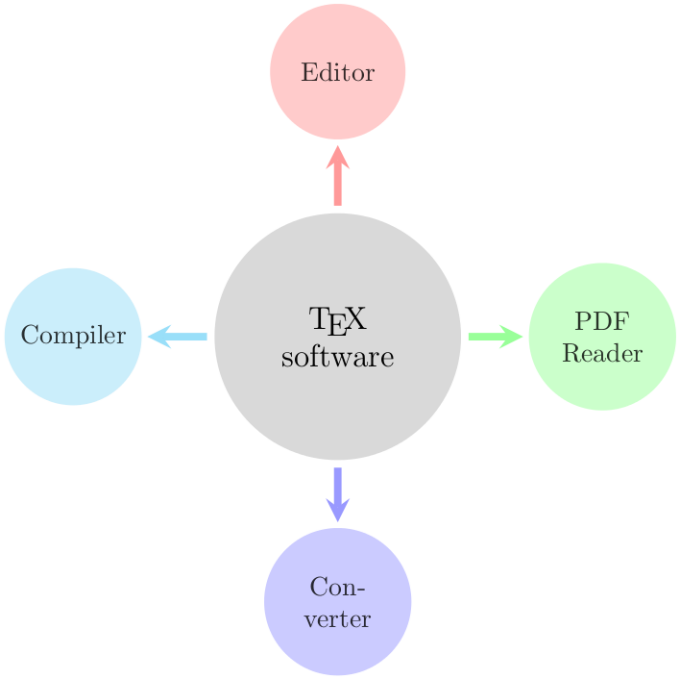
```
\smartdiagram[bubble diagram]{\TeX\ engines,
  \TeX\ (dvi), pdf\TeX, \XeTeX, \LuaTeX, \ConTeXt}
```



Constellation diagrams

Let's take a bubble diagram, and put the child concepts more far away from the center concept. Then, we point with arrows from the center to the surrounding child concept. That's what we here call a constellation diagram. Again, the center is the first item in the list:

```
\smartdiagram[constellation diagram]{\TeX\ software,
  Editor, Compiler, Converter, PDF Reader}
```

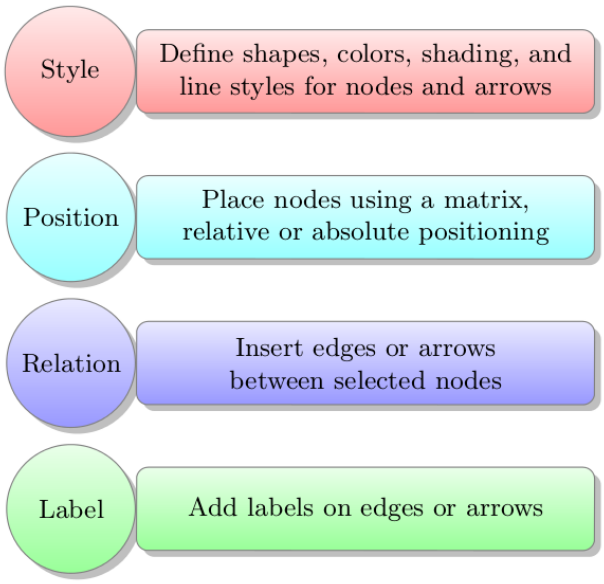


Descriptive diagrams

For nicely arranging items with a description, there's the descriptive diagram. That's a graphical list of items with a description. This requires a nested list: we need a list of lines, where each line is a small list of the item and its description. We construct them with curly braces. We use additional curly braces to hide the comma within an item so that it won't be taken as an item separator:

```
\smartdiagram[descriptive diagram]{
  {Style,{Define shapes, colors, shading,
          and line styles for nodes and arrows}},
  {Position, {Place nodes using a matrix,
              relative or absolute positioning}},
  {Relation, Insert edges or arrows
             between selected nodes},
  {Label, Add labels on edges or arrows}}
```

This gives us the following diagram:

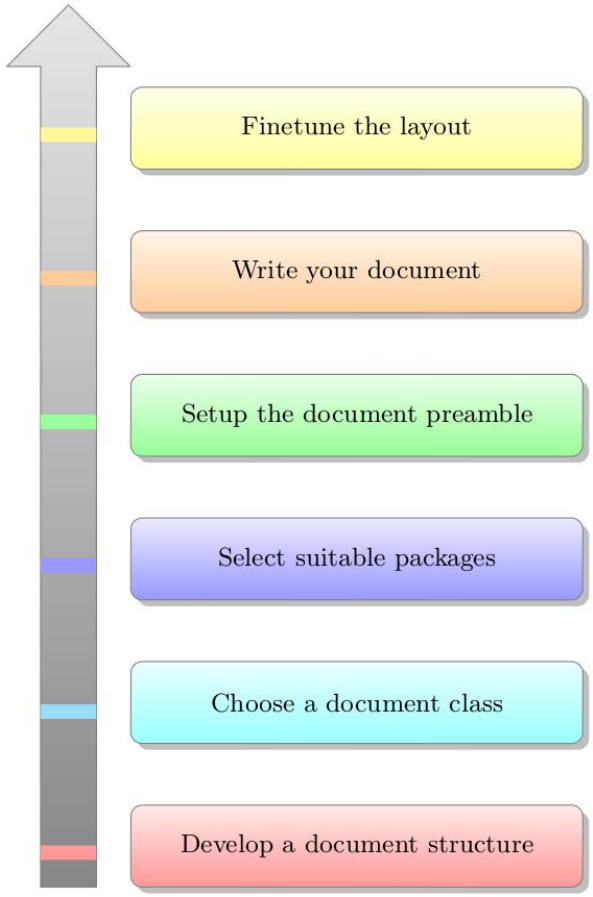


By the way, the diagram shows some tasks about drawing with TikZ, which can later be useful for us.

Priority descriptive diagrams

If your descriptive diagram has a certain order and you would like to emphasize that order, use a priority descriptive diagram, like this:

```
\smartdiagram[priority descriptive diagram]{
  Develop a document structure,
  Choose a document class,
  Select suitable packages,
  Setup the document preamble,
  Write your document,
  Finetune the layout}
```



Animating a diagram

In a presentation done with the `beamer` class, you can easily animate such a diagram. It's as simple as the following:

- Using the `beamer` class
- Using a `frame` environment for the diagram
- Writing `\smartdiagramanimated` instead of `\smartdiagram`

This, applied to one of the examples you saw earlier, results in the following:

```
\documentclass{beamer}
\usepackage{smartdiagram}
\begin{document}
\begin{frame}
  \smartdiagramanimated[circular diagram]{Edit,
  pdf\LaTeX, Bib\TeX/ biber, make\-index, pdf\LaTeX}
\end{frame}
\end{document}
```

The diagram will then be built step by step, frame by frame.

Further customization

Of course, you can set the colors and shapes of all elements. For details, refer to the `smartdiagram` manual, as usual via `texdoc`.

Constructing a flowchart

In the previous recipe, *Building smart diagrams*, we used predefined chart types. In a case when we would need more flexibility, we can do it ourselves from scratch. It's also not too hard, and it's a very good way of learning to draw with TikZ.

We will create a flowchart depicting the decision-making procedure for choosing a math environment.

Similar to what we saw in our description diagram of the first recipe, we will do the following:

1. Define shapes and colors.
2. Place nodes using a matrix.
3. Insert labeled arrows between selected nodes.

Of course, whatever is done in each step may be modified at any time. It's usual to place nodes and arrows at first, and then start fine-tuning shapes and colors.

So let's tackle it. Go through the following steps, and also look at the graphical output shown after them to understand the meaning of your drawing tasks:

1. Start with a document class:

   ```
   \documentclass{article}
   ```

2. We load the `geometry` package and specify a vertical margin so that the long chart will fit to the page:

   ```
   \usepackage[a4paper,vmargin=3cm]{geometry}
   ```

3. Load the `tikz` package:

   ```
   \usepackage{tikz}
   ```

4. Load the `matrix`, `calc`, and `shapes` TikZ libraries:

   ```
   \usetikzlibrary{matrix,calc,shapes}
   ```

5. Start the document:

   ```
   \begin{document}
   ```

6. We define styles for the nodes, which is our first major step:

   ```
   \tikzset{
     treenode/.style = {shape=rectangle, rounded corners,
                        draw, anchor=center,
                        text width=5em, align=center,
                        top color=white, bottom color=blue!20,
                        inner sep=1ex},
     decision/.style = {treenode, diamond, inner sep=0pt},
     root/.style     = {treenode, font=\Large,
                        bottom color=red!30},
     env/.style      = {treenode, font=\ttfamily\normalsize},
     finish/.style   = {root, bottom color=green!40},
     dummy/.style    = {circle,draw}
   }
   ```

7. Create some useful shortcuts for edge types:

   ```
   \newcommand{\yes}{edge node [above] {yes}}
   \newcommand{\no}{edge  node [left]  {no}}
   ```

8. Begin the document:

   ```
   \begin{document}
   ```

9. Begin the TikZ picture. Add the desired options; here, we just declare `-latex` as the default arrow tip for edges:

   ```
   \begin{tikzpicture}[-latex]
   ```

10. Now comes the second major step-positioning. Start a matrix with the name `chart`:

```
\matrix (chart)
```

11. Define the options for the matrix:

```
[
  matrix of nodes,
  column sep      = 3em,
  row sep         = 5ex,
  column 1/.style = {nodes={decision}},
  column 2/.style = {nodes={env}}
]
```

12. Now add the actual matrix contents, which will be the nodes of your flowchart. As with `tabular`, columns are separated by `&` and lines end with `\\`. This is also necessary for the last line. We modify the style of a certain node by inserting `|<style>|` before the node content. A final semicolon ends the `\matrix` command with options:

```
{
  |[root]| Formula           &                 \\
  single-line?               & equation        \\
  centered?                  & gather          \\
  aligned at relation sign?  & align, flalign  \\
  aligned at several places? & alignat         \\
  first left, centered,
    last right?              & multline        \\
  & & |[decision]| numbered? \\
  & & |[treenode]| Add a \texttt{*}
                             & |[finish]| Done \\
};
```

13. Now for the third major step-drawing arrows. Draw the **no** edges in the first column downwards and the **yes** edges to the right. We use `\foreach` loops to reduce the quantity of code. The final edge goes to the final node in the bottom-right corner:

```
\draw
  (chart-1-1) edge (chart-2-1)
  \foreach \x/\y in {2/3, 3/4, 4/5, 5/6} {
    (chart-\x-1) \no (chart-\y-1) }
  \foreach \x in {2,...,6} {
     (chart-\x-1) \yes (chart-\x-2) }
 (chart-7-3) \no  (chart-8-3)
 (chart-8-3) edge (chart-8-4);
```

14. Draw a line back to the start:

```
\draw
  (chart-6-1) -- +(-2,0) |- (chart-1-1)
    node[near start,sloped,above] {no, reconsider};
```

15. Next, draw lines from the nodes in the second column to another node down in the third column:

```
\foreach \x in {2,...,6} {
 \draw (chart-\x-2) -| (chart-7-3);}
```

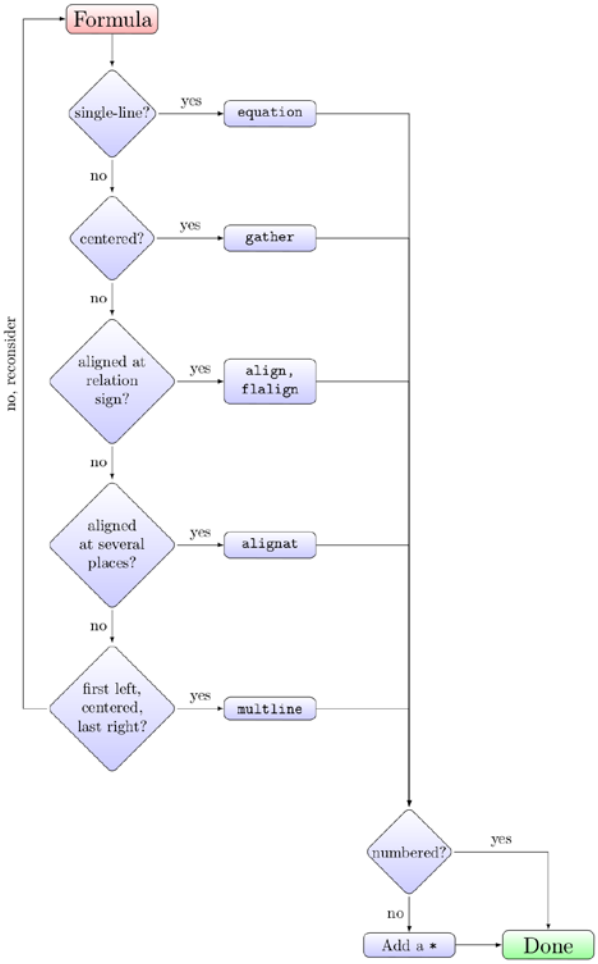
16. Draw a **yes** edge to the final node at the bottom-right corner:

```
\draw   (chart-7-3)  -| (chart-8-4)
  node[near start,above] {yes};
```

17. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

18. Compile, and examine the output:



How it works...

Positioning the nodes is one of the main things here. Providing coordinates for nodes would be too laborious and error-prone. Using relative positioning of nodes would be another option. But the easiest is defining a matrix where we can place nodes as simply as within `tabular`. This requires a kind of grid structure, however. TikZ provides a matrix library, which has a section of its own in the manual. Refer to it to understand all the options. Here, we defined only the space between rows and columns and some styles for columns.

Styles can be defined locally; for example, as an option in a `tikzpicture` environment. For repeated use, it is better to define them globally. For this, we used the `\tikzset` command. It takes a list of style assignments as the argument. As you saw the following principle:

```
thingy/.style = {list of style options}
```

You can redefine existing styles in this way, or create your own styles. Now you can simply use that new style name as an option wherever needed instead of always listing specific options. You get the same benefits here as with macros, such as readability and consistency.

You can combine styles and use them to define derived styles. So, we defined a base style for tree nodes, created additional styles that inherited it, and added some code.

Now, back to our code; our matrix structure is as follows:

```
\matrix (name) [options] {
  entry & entry & … \\
  entry & entry & … \\
  …
};
```

For additional drawing, each node of the matrix now gets a name. We can refer to it as `(name-row-column)`, just as we did here:

```
\draw (chart-1-1) edge (chart-2-1);
```

This draws a vertical edge from the top-left node of our chart to the node below it.

For column styles, we used the handy `nodes={decision}` syntax, which simply says that all nodes should have this style. We still can apply local styles, so we used the `|[style]|` shortcut at the beginning of the matrix cells.

After the matrix, we used `\draw` to draw all the edges. They are arrows in our case, since we added the `–latex` option to `tikzpicture`. We could have chosen `->` instead; `latex` gives just another arrow tip.

To repeat things efficiently, we used `\foreach` loops. The simplified syntax is as follows:

```
\foreach <variables> in {<list>} { <commands> }
```

Here, the commands can contain variables. These are replaced by list values, iterating through the whole list. The `\foreach` loop has an entire section in the TikZ manual. For our recipe, that syntax and the samples with one variable and two variables should suffice for a basic understanding.

A special edge is drawn by `-|`. This means a horizontal line and then a vertical line to the destination. The `|-` sign is its counterpart, which works the other way round. For details regarding drawing syntax and the available options, refer to the TikZ manual. It also contains tutorials that are really useful for diving into the topic quickly.

Building a tree

A very common type of hierarchical graph is a tree. Tree nodes have children. These are connected by edges and are usually displayed in rows when growing down, or in columns when growing horizontally.

We will draw a tree presenting possible math formula layouts.

How to do it...

We will use basic TikZ, without any extra package. Follow these steps:

1. Start with a document class:
   ```
   \documentclass{article}
   ```
2. Load the `tikz` package:
   ```
   \usepackage{tikz}
   ```
3. Start the document:
   ```
   \begin{document}
   ```
4. Begin with a TikZ picture, and specify some options for it:
   ```
   \begin{tikzpicture}[sibling distance=10em,
     every node/.style = {shape=rectangle, rounded corners,
       draw, align=center,
       top color=white, bottom color=blue!20}]]
   ```

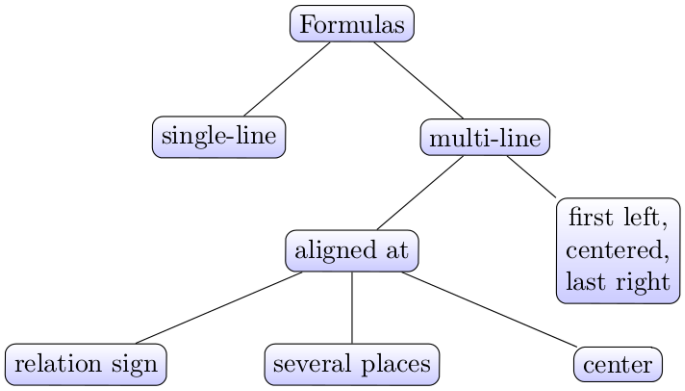
5. Draw a node and add children to it:

```
\node {Formulas}
  child { node {single-line} }
  child { node {multi-line}
    child { node {aligned at}
      child { node {relation sign} }
      child { node {several places} }
      child { node {center} } }
    child { node {first left,\\centered,\\last right} }
  };
```

6. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

7. Compile, and take a look:



How it works...

To avoid overlapping, we defined a distance between siblings. As all the nodes were the same, we defined one style for all of them, using a rounded rectangle as the drawn border and a color that transits from white to light blue. We did this by setting these options to `every node/.style`. This default style simplifies writing. We can still change specific node styles, however.

There's more...

We can play a bit with the diagram type and items to get different diagrams. If there's a root item or a central item, it's always the first in the list.

Creating a decision tree

Our recipe started with a vertical tree. We can also draw a tree horizontally. Let's do it this way now. We will use this occasion to introduce TikZ styles. A style is a set of options. Styles make drawing easier, similar to macros; we don't need to repeat the same options again and again. Instead, we refer to a desired style. Styles can also be combined.

We will now go through building the tree in detail. These are our steps:

1. Again, start with a document class:

```
\documentclass{article}
```

2. Load the `tikz` package:

```
\usepackage{tikz}
```

3. Use the `\tikzset` command to define your own new styles:

```
\tikzset{
  treenode/.style = {shape=rectangle, rounded corners,
                     draw, align=center,
                     top color=white,
                     bottom color=blue!20},
  root/.style     = {treenode, font=\Large,
                     bottom color=red!30},
  env/.style      = {treenode, font=\ttfamily\normalsize},
  dummy/.style    = {circle,draw}
}
```

4. Start the document:

```
\begin{document}
```

5. Begin with the TikZ picture:

```
\begin{tikzpicture}
```

6. Add options to the TikZ picture. As before, use square brackets:

```
[
  grow                    = right,
  sibling distance        = 6em,
  level distance          = 10em,
  edge from parent/.style = {draw, -latex},
  every node/.style       = {font=\footnotesize},
  sloped
]
```

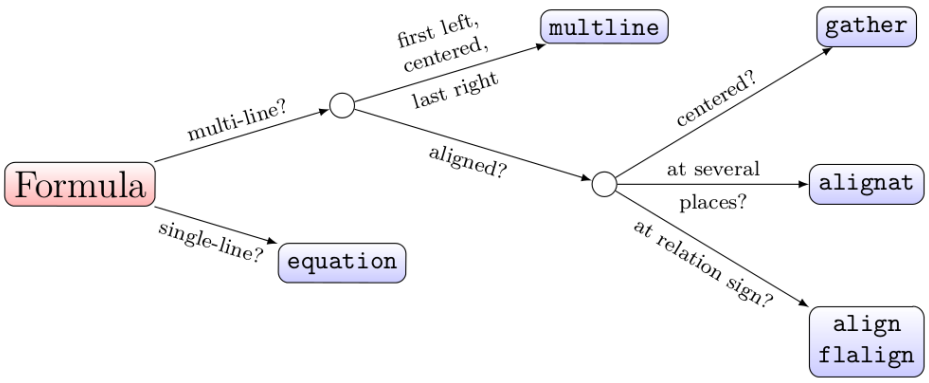
7. Declare the nodes and children in the tree hierarchy. If an edge needs to be labeled, add a node using the `edge from parent` option after the node:

```
\node [root] {Formula}
  child { node [env] {equation}
    edge from parent node [below] {single-line?} }
  child { node [dummy] {}
    child { node [dummy] {}
      child { node [env] {align\\flalign}
        edge from parent node [below]
                          {at relation sign?} }
      child { node [env] {alignat}
        edge from parent node [above] {at several}
                          node [below] {places?} }
      child { node [env] {gather}
              edge from parent node [above] {centered?} }
      edge from parent node [below] {aligned?} }
    child { node [env] {multline}
            edge from parent node [above, align=center]
              {first left,\\centered,}
            node [below] {last right}}
            edge from parent node [above] {multi-line?}};
```

8. End the TikZ picture and the document:

```
\end{tikzpicture}
\end{document}
```

9. Compile, and take a look:



The main difference is `grow=right`, for getting a horizontal tree. Besides this, we used various custom styles for the nodes. After a node that is connected to its parent by an edge, we use the `edge from parent` option to add a node with label text.

Building a bar chart

A classic way of displaying categories with corresponding values is a bar chart. It consists of rectangular bars, and they are proportional to the represented values. The main purpose is to visually compare those values.

We will draw a chart visualizing web forum statistics.

How to do it...

We will use the `pgfplots` package. It's for natively plotting in LaTeX with a convenient user interface. We will use it to produce a horizontal bar chart.

Perform these steps:

1. Start with a document class:

```
\documentclass{article}
```

2. Load the `pgfplots` package:

```
\usepackage{pgfplots}
\pgfplotsset{width=7cm,compat=1.8}
```

3. Begin the document:

```
\begin{document}
```

4. Begin a TikZ picture, which will be the container for the plot:

```
\begin{tikzpicture}
```

5. Open an axis environment:

```
  \begin{axis}
```

6. Give options to the axis:

```
 [
    title    = Contributions per category
               at LaTeX-Community.org,
    xbar,
    y axis line style = { opacity = 0 },
    axis x line       = none,
    tickwidth         = 0pt,
    enlarge y limits  = 0.2,
    enlarge x limits  = 0.02,
    nodes near coords,
    symbolic y coords = {LaTeX, Tools,
                         Distributions, Editors},
  ]
```

7. Add a plot:

```
  \addplot coordinates { (57727,LaTeX) (5672,Tools)
           (2193,Distributions) (11106,Editors) };
```

8. Add another plot:

```
  \addplot coordinates { (14320,LaTeX) (1615,Tools)
           (560,Distributions)  (3075,Editors)  };
```

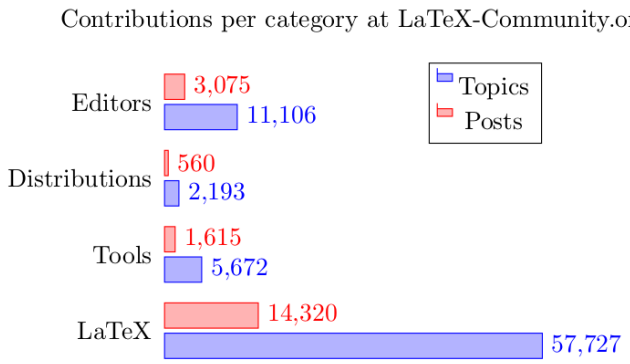
9. Then, add a legend:

```
  \legend{Topics, Posts}
```

10. End the `axis` environment, the TikZ picture, and the whole document:

```
  \end{axis}
\end{tikzpicture}
\end{document}
```

11. Compile, and take a look:



How it works...

The `\addplot` command does the plotting work; we called it with a set of specific coordinates. In our case, we used numeric values together with symbolic coordinates for the subjects.

We designed the plot in Step 6 by adding a lot of customization options to the `axis` environment. This is the specific `pgfplots` environment with a lot of options for customizing. Here's what we did:

1. We provided a title.
2. We set `xbar` as a style to get bars in the *x* direction. Specifically, this means that horizontal bars will be placed from *y=0* to the *x* coordinate.
3. Omitting unnecessary diagram parts means less distraction. This helps in focusing and understanding the contents. Therefore, we did the following:
   - We chose a completely opaque line style for the *y* axis so that it would n't be printed
   - We hid the x axis using `axis x line = none`
   - We removed the *x* axis ticks by setting their width to `0`
4. Then, we enlarged the `x` and `y` limits a bit to get a better display.
5. We placed the nodes with values near the bars by enabling the `nodes near coords` option.
6. We defined symbolic coordinates as `y` values so that we could assign numeric `(x)` values to `(y)` subjects.

To sum up, `xbar` defined the plot style, and `symbolic y coords` let us use string values. The other axis options were just for the design. There are many more possible settings, so for further customization, refer to the `pgfplots` manual. You can open it by entering `texdoc pgfplots` in Command Prompt or visiting `http://texdoc.net/pkg/pgfplots`.

Drawing a pie chart

Pie charts are popular for showing proportions. A pie chart's main characteristic is that all items usually sum up to 100 percent. They are displayed as segments of a disc.

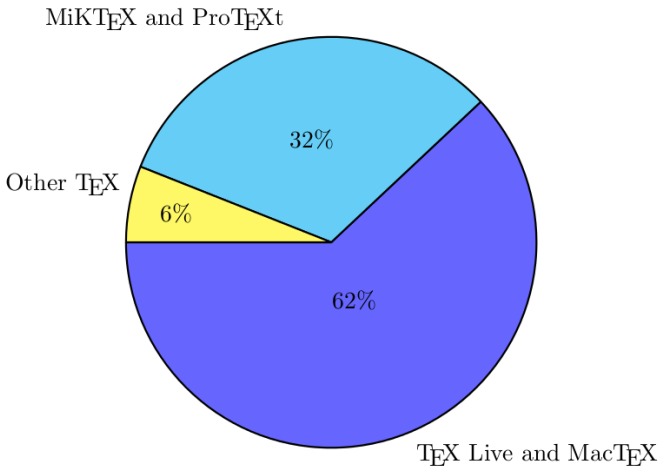
As an example, we will use a pie chart to display the relative amount of questions about different TeX distributions.

How to do it...

We will use the `pgf-pie` package, which builds on TikZ and is specialized for generating pie charts. Follow these steps:

1. Start with a document class:
   ```
   \documentclass{article}
   ```
2. Load the `pgf-pie` package:
   ```
   \usepackage{pgf-pie}
   ```
3. Begin the document:
   ```
   \begin{document}
   ```
4. Begin a TikZ picture, which will be the container of the pie chart:
   ```
   \begin{tikzpicture}
   ```
5. Draw a pie chart using this command:
   ```
   \pie [rotate = 180]
      {62/\TeX\ Live and Mac\TeX,
       32/MiK\TeX\ and Pro\TeX t, 6/Other \TeX}
   ```
6. End the TikZ picture and the entire document:
   ```
   \end{tikzpicture}
   \end{document}
   ```

7. Compile and take a look at the result:



How it works...

The `\pie` command is the only user command in the `pgf-pie` package. The syntax is as follows:

```
\pie[options]{number1/text1, number2/text2, …}
```

The backslashes in our example were just because of the `\TeX` macro and the following space.

Let's take a look at the available options, with example values:

- `pos = 4,6`: This positions the center at point `(4,6)`. The default center is `(0,0)`.
- `rotate = 90`: This rotates the chart by `90` degrees.
- `radius = 5`: This sets the chart's radius size to `5`. The default size is `3`.
- `color = red`: This chooses the color red for all the slices. We can use any color syntax which TikZ understands, such as `red!80!black` for a mix of 80% red and 20% black.
- `color = {red!20, red!40, red!60}`: This sets a specific red color value for each of the three slices.
- `explode = 0.1`: This moves all the slices outwards by `0.1`.
- `explode = { 0.2, 0, 0}`: This moves only the first slice of the three outwards by `0.2`.
- `sum = 50`: This defines the reference sum as `50`, instead of the default sum of `100`.

- `sum = auto`: This calculates the sum from the slice values.
- `scale font`: This scales the font size according to the slice value.
- `before number = { \$ }`: This inserts text before the values, in this case a dollar sign. It is empty by default.
- `after number = { percent }`: This adds text after each value, in this case the word `percent`. With `sum = 100`, the default is %; otherwise, it is empty.
- `text = pin`: This sets the text next to the slice, connected by a short line.
- `text = inside`: This places the text within the slice.
- `text = legend`: This produces a separate legend.
- `style = drop shadow`: This adds a shadow below the chart.

There's more...

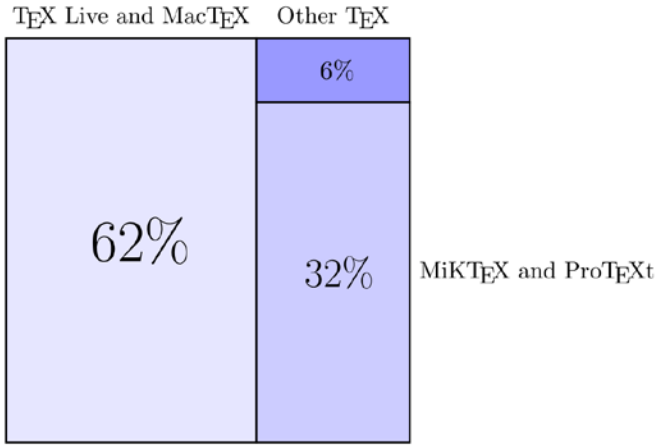
The `pie-chart` package offers further chart designs. Let's take a look at them, together with applying some of the styles we just covered.

Square charts

The `square` option gives a quadratic design. Adding the `scale font` and `color` options, we arrive at this:

```
\pie [square, scale font,
      color = {blue!10, blue!20, blue!40}] { ... }
```

With the values from our recipe, we get the following:

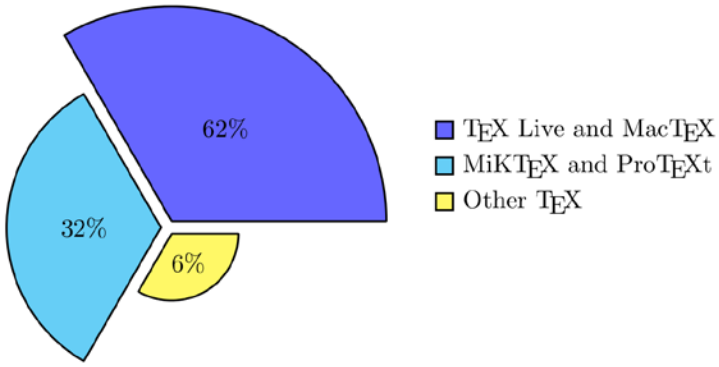


Polar area charts

The `polar` option changes the layout so that the slices get equal angles but the radius represents the size. We add the `explode` and `text=legend` options:

```
\pie [polar, explode=0.1, text=legend] { ... }
```

Thus, we get this output:

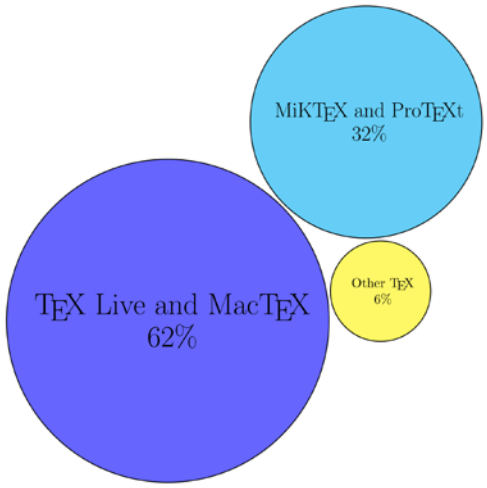


Cloud charts

The `cloud` option produces a set of discs whose sizes are determined by the given values. This time, we put the text inside, scale it, and use a larger radius:

```
\pie [cloud, text=inside, scale font, radius=6] { ... }
```

Now, the result is as follows:



Drawing a Venn diagram

A Venn diagram displays several sets with their relationships. Commonly, these are overlapping circles. Such sets can stand for certain properties. If an element has two such properties, it will belong to an overlapping area-the intersection of the two relevant sets.

In this recipe, we will draw a Venn diagram of three sets.

How to do it...

We will draw colored circles and apply blending to their intersections. Go through these steps:

1. Choose a document class:

```
\documentclass{article}
```

2. Load the `tikz` package:

```
\usepackage{tikz}
```

3. Begin the document:

```
\begin{document}
```

4. Begin a TikZ picture environment:

```
\begin{tikzpicture}
```

5. Use a `scope` environment to apply a style to a part of the drawing. Here, we apply color blending:

```
  \begin{scope}[blend group=soft light]
```

6. Draw the diagram parts, which in our case are simply filled circles:

```
    \fill[red!30!white]   ( 90:1.2) circle (2);
    \fill[green!30!white] (210:1.2) circle (2);
    \fill[blue!30!white]  (330:1.2) circle (2);
```

7. End the `scope` environment. At the end of the environment, the blending effect will end, because environments keep the settings local:

```
  \end{scope}
```

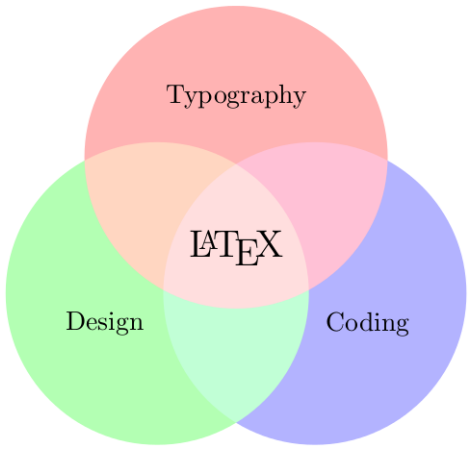
8. Add nodes with text for the descriptions:

```
  \node at ( 90:2)    {Typography};
  \node at (210:2)    {Design};
  \node at (330:2)    {Coding};
  \node [font=\Large] {\LaTeX};
```

9. End the TikZ picture and the document:

```
\end{tikzpicture}
\end{document}
```

10. Compile, and take a look at the output:



How it works...

We created three filled circles. The center of each circle is specified in polar coordinates, with a given angle and distance from the origin. This makes radial placement easier. For example, the first circle has its center at (90:1.2). This means that the center is at 90 degrees, which is above the origin, and the distance to the origin is 1.2. The radius of each circle is 2, so they are overlapping.

Normally, overlapping would simply mean that the final circle overrides what is below it. We still wish to look behind the circles to see the intersections. A classical approach is to use transparency, like this, for example:

```
\begin{scope}[opacity=0.5]
   ...
\end{scope}
```

This lets the background shine through. We used a `scope` environment to keep the setting local. The `opacity` value can be between 0, which means totally transparent, and 1, which means totally opaque.

Another pleasing way for getting transparent graphics is by using the **blend mode** feature of the PDF standard; that's what we did. It's mixing colors in a certain way. The possible ways are as follows, in short:

- `normal`: Any object is simply drawn over the background.
- `multiply`: Color values of mixed objects are multiplied. A black factor always produces black and a white factor causes no change. Generally, we get darker colors as the product.
- `screen`: This complements the color values, multiplies, and complements again. A white factor always produces white and a black factor causes no change. Generally, we get lighter colors in such a mix.
- `overlay`: This is like the multiply or the screen option. It depends on the background color.
- `darken`: The darker of the mixed colors is chosen.
- `lighten`: The lighter color of the mix is chosen.
- `colordodge`: The background is brightened to reflect the foreground.
- `colorburn`: The background is darkened to reflect the foreground.
- `hardlight`: This is like the multiply or the screen option. It depends on the foreground color. It appears like a hard spotlight.
- `softlight`: This darkens or lightens the color, depending on the foreground color, like a softened spotlight.
- `difference`: This subtracts the darker color from the lighter color.
- `exclusion`: This is like difference, but with lower contrast.
- `hue`: The resulting color has the hue of the foreground and the saturation and luminosity of the background.
- `saturation`: The resulting color has the saturation of the foreground and the hue and luminosity of the background.
- `color`: The resulting color has the hue of the foreground and the saturation and luminosity of the background.
- `luminosity`: This is the inverse of `color`; the resulting color has the luminosity of the foreground and the hue and saturation of the background.

These modes are described in the PDF standard, which is cited in the TikZ manual, in the *Blend modes* section.

What seems a bit ambitious just for overlapping colors in a Venn diagram could be generally useful in overlapping drawings. You may try out those modes to see what fits your needs best.

Putting thoughts in a mind map

A mind map visualizes information or ideas. Usually, there's a main concept in the center and major concepts branching outward from it. Smaller ideas start from the major concepts, so a mind map can look like a spider web.

In this recipe, we will draw a mind map of the concepts of TeX.

How to do it...

We will use the TikZ `mindmap` library. Follow these steps:

1. Start with a document class:
   ```
   \documentclass{article}
   ```
2. Load the `geometry` package with the `landscape` option so that your wide mindmap will fit the page:
   ```
   \usepackage[landscape]{geometry}
   ```
3. Then, load the `tikz` package:
   ```
   \usepackage{tikz}
   ```
4. Next, load the `mindmap` TikZ library:
   ```
   \usetikzlibrary{mindmap}
   ```
5. Load the `dtklogos` package to get additional TeX-related logo macros:
   ```
   \usepackage{dtklogos}
   ```
6. Start the document and begin the TikZ picture:
   ```
   \begin{document}
   \begin{tikzpicture}
   ```
7. Start a path with options:
   ```
     \path [
   ```
8. Provide the `mindmap` option and choose the white text color:
   ```
       mindmap,
       text = white,
   ```

9. Adjust the styles for the levels of the map:

```
level 1 concept/.append style =
  {font=\Large\bfseries, sibling angle=90},
level 2 concept/.append style =
  {font=\normalsize\bfseries},
level 3 concept/.append style =
  {font=\small\bfseries},
```

10. Define some styles for use in related concepts, and end the option list:

```
tex/.style     = {concept, ball color=blue,
  font=\Huge\bfseries},
engines/.style = {concept, ball color=green!50!black},
formats/.style = {concept, ball color=blue!50!black},
systems/.style = {concept, ball color=red!90!black},
editors/.style = {concept, ball color=orange!90!black}
]
```

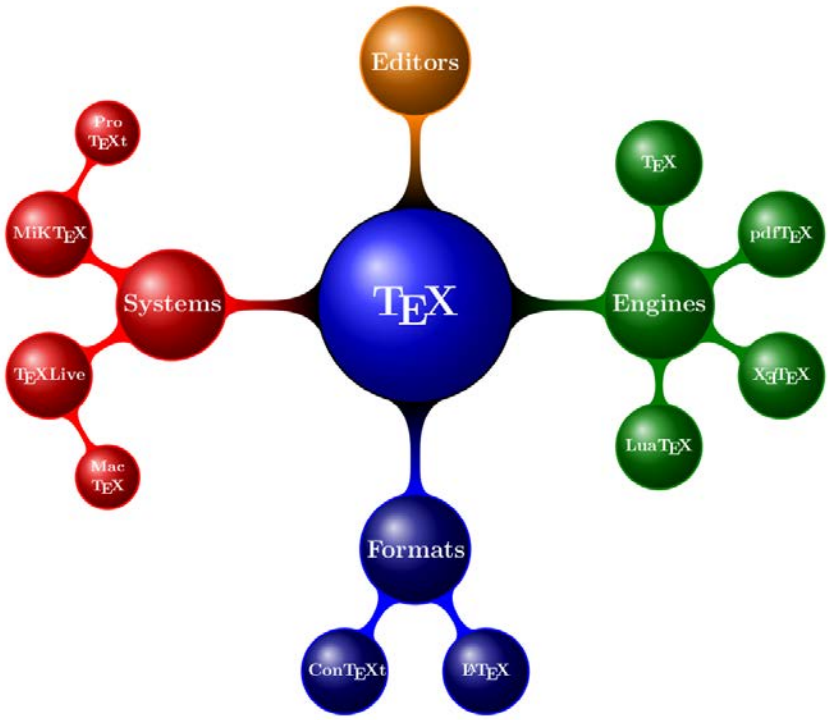
11. Now place the main concept node and its children. The structure is similar to the *Building a tree recipe* in this chapter. End the path with a semicolon:

```
node [tex] {\TeX} [clockwise from=0]
  child[concept color=green!50!black, nodes={engines}] {
    node {Engines} [clockwise from=90]
      child { node {\TeX} }
      child { node {pdf\TeX} }
      child { node {\XeTeX} }
      child { node {Lua\TeX} }}
  child [concept color=blue, nodes={formats}] {
    node {Formats} [clockwise from=300]
      child { node {\LaTeX} }
      child { node {\ConTeXt} }}
  child [concept color=red, nodes={systems}] {
    node {Systems} [clockwise from=210]
      child { node {\TeX Live} [clockwise from=300]
        child { node {Mac \TeX} }}
      child { node {MiK\TeX} [clockwise from=60]
        child { node {Pro \TeX t} }}}
  child [concept color=orange, nodes={editors}] {
    node {Editors} };
```

12. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

13. Compile, and examine the output:



How it works...

The most important option here is the `mindmap` style, which adds further styles and settings for implicit use, such as the `concept` style for nodes. They are responsible for the basic appearance.

First, we fine-tuned the design by adding style options for each level of subconcept. So, we chose a bold font-it improved the readability of the white text in darker concepts-and chose smaller font sizes for concepts farther away from the root. After doing this, we added a `sibling angle` value to tell TikZ the angle between major concepts. We would have chosen a smaller angle if we had more concepts.

Furthermore, we created our own styles. They extend the standard concept node style by adding ball shading. This gives us a 3D appearance. On one hand, it should be just a demonstration of how to add some bells and whistles; on the other hand, flat colors may seem a bit boring if we remove the ball color option to get a more sober style. You may try colors and shadings offered by TikZ.

The path is just like the tree in our earlier tree recipe. The notable options are as follows:

- `start angle`: This is set by `clockwise from = ...`, which could be `counterclockwise` as an alternative.
- `nodes = {style name}`: This applies our chosen style to all children.
- `concept color`: This is for the color behind our ball shading. It is also applied to the node connections.

The `mindmap` library has a dedicated section in the TikZ manual. There, you can read about alternative connections between nodes, and adding annotations.

Generating a timeline

By using a basic TikZ function, it's not too hard to create a line and add some ticks, date values, and annotations. But it can require a lot of work, until it looks nice.

This recipe will show you another colorful and predesigned way: using the `timeline` library.

Getting ready

Until the `timelime` library officially becomes part of TikZ or becomes available on CTAN, you can download the `tikzlibrarytimeline.code.tex` file from its author's repository at `https://github.com/cfiandra/timeline`. You can install it in the TeX tree like any other package, but the easiest way is to simply put it into the same folder as your main TeX document.

How to do it...

The `timelime` library builds on TikZ, but also provides its own high-level commands. Follow these steps:

1. Start with a document class whose paper size is big enough:

```
\documentclass[a3paper]{article}
```

2. Load the `geometry` package with the `landscape` option, since your diagram will be wider than it is high:

```
\usepackage[landscape]{geometry}
```

3. Load the `tikz` package:

```
\usepackage{tikz}
```

4. Then, load the `timeline` TikZ library:

```
\usetikzlibrary{timeline}
```

5. Start the document and begin the TikZ picture:

```
\begin{document}
\begin{tikzpicture}
```

6. State the number of weeks:

```
\timeline{5}
```

7. Define the time phases:

```
\begin{phases}
  \initialphase{involvement degree=3cm,phase color=blue}
  \phase{between week=1 and 2 in 0.4,
    involvement degree=5cm,phase color=green!50!black}
  \phase{between week=2 and 3 in 0.2,
    involvement degree=6cm,phase color=red!40!black}
  \phase{between week=3 and 4 in 0.5,
    involvement degree=3cm,phase color=red!90!black}
  \phase{between week=4 and 5 in 0.3,
    involvement degree=2.5cm,phase color=red!40!yellow}
\end{phases}
```

8. Add some text nodes as annotations on the left-hand side:

```
\node [xshift=-0.6cm,yshift=1cm,anchor=east,
       font=\Large\bfseries] at (phase-0.180)
       {Auhor};
\node [xshift=-0.6cm,yshift=-1cm,anchor=east,
       font=\Large\bfseries] at (phase-0.180)
       {Publisher};
```

9. Add milestones on the upper side of the timeline:

```
\addmilestone{at=phase-0.120,direction=120:1cm,
  text={Concept}, text options={above}}
\addmilestone{at=phase-0.90,direction=90:1.2cm,
  text={Outline}}
\addmilestone{at=phase-1.110,direction=110:1.5cm,
  text={Research}}
```

```
\addmilestone{at=phase-2.100,direction=100:1cm,
  text={Writing}}
\addmilestone{at=phase-2.60,direction=90:1.5cm,
   text={First draft}}
\addmilestone{at=phase-3.90,direction=90:1.2cm,
  text={Second draft}}
 \addmilestone{at=phase-4.90,direction=90:0.8cm,
   text={Approval of print draft}}
```

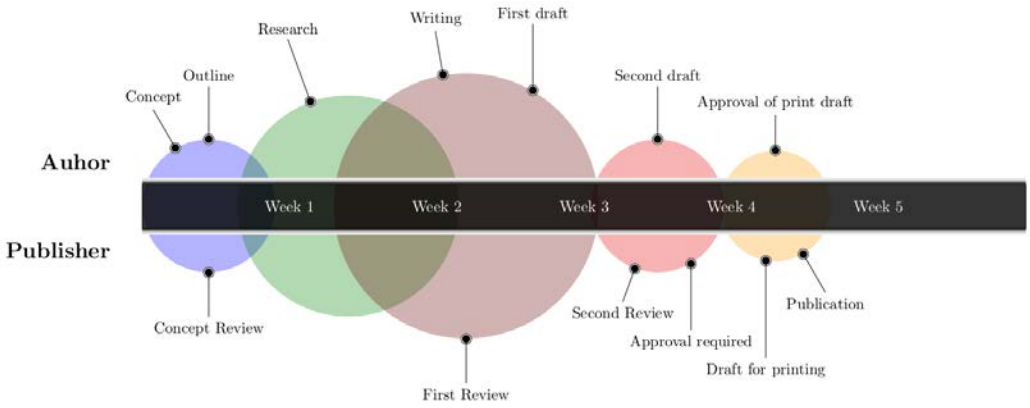
10. Then, add milestones on the lower side of the timeline:

```
\addmilestone{at=phase-0.270,direction=270:1cm,
  text={Concept Review}, text options={below}}
\addmilestone{at=phase-2.270,direction=270:1cm,
 text={First Review}}
\addmilestone{at=phase-3.250,direction=250:0.8cm,
  text={Second Review}}
\addmilestone{at=phase-3.300,direction=270:1.5cm,
  text={Approval required}}
\addmilestone{at=phase-4.260,direction=270:2.2cm,
 text={Draft for printing}}
\addmilestone{at=phase-4.300,direction=300:1cm,
  text={Publication}}
```

11. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

12. Compile, and examine the output:



How it works...

With `\timeline{5}`, we created a filled rectangle with five weeks. Choose another number for more or fewer weeks. There's a node for each week that you can use for additional drawing; simply refer to `(week-1)` and so on as named nodes.

Within the `phases` environment, we defined the filled circles. They stand for the various time phases. Let's look at the sample phase arguments to understand their use:

- `between week=1 and 2 in 0.4`: This means starting at week 1 and ending at week 2, with an offset of 0.4 for fine-tuning
- `involvement degree`: This is the radius of the phase circle
- `phase color`: This is the fill color of the phase circle

That's specific syntax for basic elements as such.

Finally, we added milestones. These are text nodes connected to a phase with a line:

- `At`: This means the starting position. It's good to use relative positioning, such as `phase-1.north` for "above the phase-1 node". We used the `phase-n.angle` TikZ syntax here.
- `direction`: This is given as polar values, like this: `(angle:distance)`.
- `text`: This is the text within the node. We used the `text options` key to customize the placement. This option is sticky, so it will be remembered. The text was placed `above` until we changed `text options` to `below`, which was then kept.

The library gives a nice quick start for a project plan.

10
Advanced Mathematics

This chapter covers the following topics:

- Quick-start for beginners
- Fine-tuning a formula
- Automatic line-breaking in equations
- Highlighting in a formula
- Writing theorems and definitions
- Drawing a commutative diagram
- Plotting functions in two dimensions
- Plotting in three dimensions
- Drawing geometry pictures
- Doing calculations

Introduction

One of the classic strong points of LaTeX is its excellent quality in typesetting formulas. That's why LaTeX is the most used writing software in mathematics. Other sciences benefit from it as well when it comes to formulas.

This chapter provides selected recipes for common tasks. It's expected that readers already know the basics of writing math but to also support beginners we will start with a quick tutorial. If you already have math writing experience, you can safely skip the first recipe.

Like in the previous chapter, we will see some recipes for creating graphics. This time we will focus on mathematics, such as visualizing maps and functions, and practicing geometry.

At the end you can find a list with documentation and online resources for mathematics with LaTeX.

Quick-start for beginners

LaTeX provides a special syntax for math expressions. Furthermore, it has to be declared what is math but not text to get properly printed. Let's see how to write math!

How to do it...

We will practice math basics now. Let's write a document:

1. As usual, start with a document class.

```
\documentclass{article}
```

2. Start the document:

```
\begin{document}
```

3. Let's have an unnumbered section for our text:

```
\section*{The golden ratio}
```

4. Write a statement containing math. Enclose each math formula within parentheses. As usual for LaTeX commands, write a backslash before each parenthesis.

```
The symbol for the golden ratio is the Greek
letter \( \varphi \). Its value is the positive
solution of \( x^2 - x - 1 = 0 \).
```

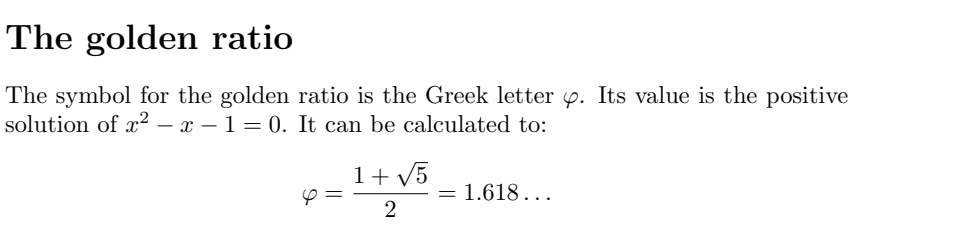
5. Continue the text; this time, math shall be set off and centered:

```
It can be calculated to:
\[
  \varphi = \frac{1 + \sqrt{5}}{2} = 1.618 \ldots
\]
```

6. End the document:

```
\end{document}
```

7. Compile the document, and take a look:



The golden ratio

The symbol for the golden ratio is the Greek letter φ. Its value is the positive solution of $x^2 - x - 1 = 0$. It can be calculated to:

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.618\ldots$$

How it works...

Math is differently written compared to normal text. For example, letters are printed in italic font to distinguish variables from normal text. That's why we need to tell LaTeX when the so-called **math mode** starts and when it ends, even for a single symbol.

Math styles

There are two basic styles:

- **Inline math**: A formula is embedded within normal text. Write it this way:

  ```
  text \( formula \) more text
  ```

- **Displayed math**: If the formula should be emphasized, or if it's so big that it's more readable in its own paragraph, use square brackets instead of parentheses, to center it with space before and after:

  ```
  \[ formula \]
  ```

  Such a displayed formula will be centered, and there's space before and after it. There should not be empty lines in the code before or after, as it would add paragraph breaks.



There's the older TeX syntax `$ ... $` for inline math, which works in LaTeX as well. It's shorter and still popular. However, the also existing TeX syntax `$$ ... $$` for displayed math should not be used as the vertical spacing would not be consistent with LaTeX.

Greek letters

You noticed the Greek letter phi (φ) in the preceding formula. Generally, the command for a Greek letter is like its name with a backslash. So there are `\alpha`, `\beta`, `\gamma`, `\delta`, and so on, for lowercase Greek letters, and `\Gamma`, `\Delta`, and so on, for uppercase Greek letters. For individual Greek letters, which look like Latin letters, there's no standard command as we could simply type them directly.

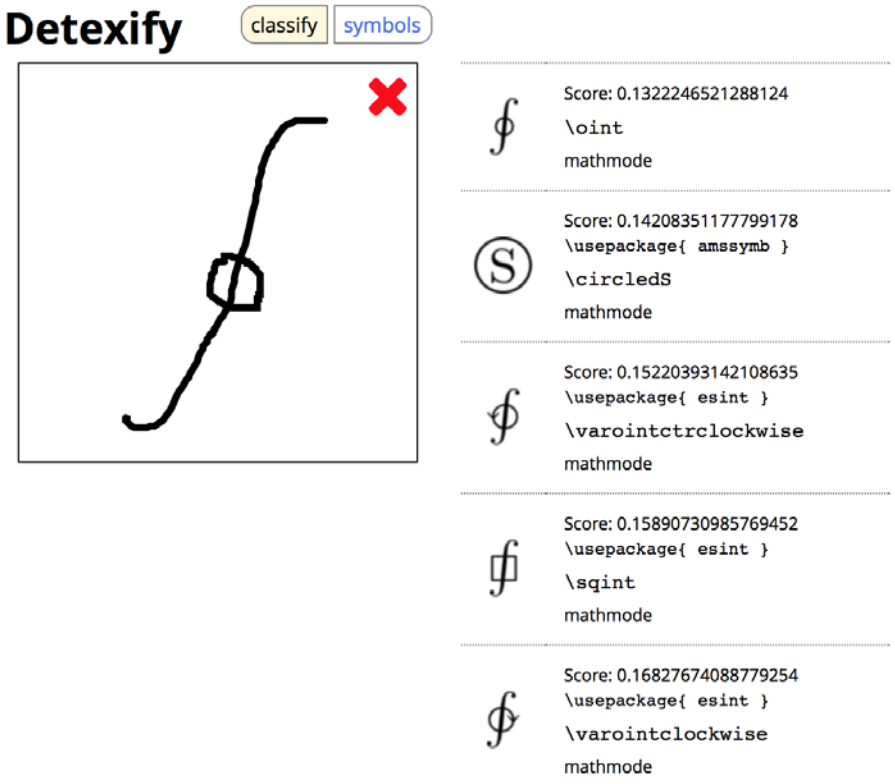
Math symbols

Mathematicians are ingenious. They invented a lot of math symbols, such as for operations and relations. Generations of mathematicians have used LaTeX, so nearly any math symbol we can think of is provided by LaTeX or an additional package, such as `latexsym` or `amssymb`. Load such a package as usual via the `\usepackage` command.

However, you need to know the command for the symbol. There's a big list of more than 5,000 symbols. Luckily, it's sorted by topic and provides clear tables with explanations. You can open it on your computer by typing the `texdoc symbols` command in Command Prompt or visit it online at `http://texdoc.net/pkg/symbols`.

Searching in that list can be a bit time-consuming though. The **detexify** tool comes to the rescue. It's a clever online tool that takes hand-drawn symbols, such as those drawn using the mouse or touch screen, and gives you in return LaTeX commands that produce such a symbol. You can find it here: `http://detexify.kirelabs.org`.

Here you can see how it works on my badly hand-drawn surface integral symbol:



It shows you the LaTeX command, its expected output, and the required package. It is sorted, so the best match is on top, like in our test case.

Detexify is also available as an app for the iPhone, iPad, and Android devices.

Squares and fractions

You saw further LaTeX commands in our first example:

- The `\sqrt{expression}` command gives a square root of the expression. There's an optional argument for other roots. For example, type the `\sqrt[3]{x}` command to denote the third root of *x*.
- The `\frac{numerator}{denominator}` command prints a fraction, with the numerator above the denominator, separated by a line.



Be spacey! LaTeX ignores spaces in formulas. So you can insert as many spaces as you like, which can improve the readability of complex formulas.

There's more...

We need a lot more for writing formulas. Let's have a quick look at some syntax.

Subscripts and superscripts

A subscript, like an index, can be attached by adding it using an underscore. For example, x_1 can be written as `x_1`. A superscript, like an exponent, is attached using a caret. So, x^2 is written `x^2`.

If a subscript or superscript should contain more than one character, group them by curly braces, such as in `x_{ij}` and `y^{12}`.

You can combine subscripts and superscripts by simply lining them up. You can nest them. But use curly braces to clearly group what belongs together. For example, this code shows different meanings just because of braces:

```
\[
  x^{n_1} \neq x^n_1
\]
```

On the left-hand side, x has an exponent n_1. On the right side, x_1 has an exponent n:

$$x^{n_1} \neq x_1^n$$

Even an expression that has the same mathematical meaning can show a difference in print:

```
\[
  {x^2}^3 = x^{2^3}
\]
```

On the left-hand side, you have two exponents. On the right-hand side, one exponent has another exponent, which is printed in smaller size:

$$ {x^2}^3 = x^{2^3} $$

So, you may use braces to define the actual meaning. You even have to use braces in case of multiple superscripts or multiple subscripts otherwise LaTeX would raise an error.

Operators

Commonly, variables are written in italics. Math functions, also called **operators**, are usually written in upright Roman letters because of their different meaning. Many operators are predefined, so you can use them just with a backslash, such as `\lim`, `\sin`, `\cos`, `\log`, `\min`, `\max`, and many more. If you notice a missing operator, you can define it yourself. Load the `amsmath` package for this, and ensure that your preamble contains this:

```
\usepackage{amsmath}
```

Then, you can define a new operator "`diff`" in your preamble by typing the following command:

```
\DeclareMathOperator{\diff}{diff}
```

You will notice that subscripts or superscripts of operators in displayed equations can be written below or above them, respectively. That's the case with the `\lim` command, for example. You can achieve the same for your new operator if you add a star as follows:

```
\DeclareMathOperator*{\diff}{diff}
```

We will use our own operators in one of the next recipes, *Drawing commutative diagrams*.

Numbering and referencing equations

Displayed formulas can be numbered automatically for cross-referencing as follows:

1. Start an `equation` environment:

   ```
   \begin{equation}
   ```

2. Give it a label:

   ```
   \label{parabola}
   ```

3. Write the formula as follows:

   ```
   y = x^2 + 1
   ```

   The formulas or equations would get an automatically incremented number in parentheses at the right, such as (1).

4. End the `equation` environment:

   ```
   \end{equation}
   ```

5. Now, you can reference it anywhere in the whole document by typing this:

   ```
   See formula (\ref{parabola}).
   ```

Note that we wrote the parentheses ourselves. If you load the `amsmath` package, which is recommended anyway, you can use its command to print the reference together with parentheses:

```
See formula \eqref{parabola}.
```



To resolve references, LaTeX needs a second compiler run. The first one writes the label to the external `.aux` file so that the second run can find it there.

Writing multiline formulas with alignment

Now you really need to load the `amsmath` package, which is definitely a must-have for writing mathematics with LaTeX. Add this to your preamble:

```
\usepackage{amsmath}
```

Now, you have several options.

Aligning at the relation symbol

Here's how to align equations the relation symbol:

- Use the `align` environment
- Mark the relation symbol for alignment by placing an ampersand (`&`) right before
- Break lines by using `\\`

So, your code will look similar to this:

```
\begin{align}
  y &= x^2 + 1 \\
  z &= 0
\end{align}
```

Now the equations are properly aligned at the equation sign:

$$y = x^2 + 1 \tag{1}$$

$$z = 0 \tag{2}$$



On the Internet and in old documents, you still can see the `eqnarray` environment for such a purpose. Don't use it as the spacing around the relation sign would be wrong, compared to simple equations.

Centering a block of equations

It's a bit simpler than the preceding example:

- Use the `gather` environment
- Break lines by using `\\`

So, your code will now look similar to this:

```
\begin{gather}
  y = x^2 + 1 \\
  z = 0
\end{gather}
```

Now the lines are aligned with each other.

Adjusting numbering

The `equation`, `align`, `gather`, and further `amsmath` environments are numbered by default. You can switch off the numbering by using their starred versions, such as `equation*` (`\[ ... \]` in standard LaTeX), `align*`, `gather*`, and so on.

Instead, you can suppress numbering for specific lines by adding the `\nonumber` command at the end of a line:

```
\begin{gather}
  y = x^2 + 1 \nonumber \\
  z = 0
\end{gather}
```

As written before, you can add labels for referencing in each line you want. Note though that having a number as tag is only meaningful when you also add a reference to it. Tags are for referencing, not for counting.



Instead of having numbers, you can set your own tags, such as (`*`), by inserting the `\tag{*}` command on a formula line.

The package `mathtools` helps in this regard. It shows equation numbers and tags only if there's a reference by the `\eqref` or `\refeq` commands (the `mathtools` package reference command version). Activate it by using this code:

```
\usepackage{mathtools}
\mathtoolsset{showonlyrefs,showmanualtags}
```

This package offers much more, which you can see in the next recipe.

Fine-tuning math formulas

Though LaTeX's typesetting of formulas is usually excellent, there are situations when the layout can require improvement. This recipe shows quick fixes.

Getting ready

Let's have a look at misalignment and spacing, issues with subscript and superscript that can easily occur:

1. Open the following sample document with your LaTeX editor. You can take it from the code archive provided with the book or copy and paste from the e-book:

```
\documentclass{article}
\usepackage{dsfont}
\begin{document}
\[
  \lim_{n\to\infty} \sup_{x\in\mathds{R}} f_n(x^2)
    < n \Big(\sum_{x\in\mathds{R}, n\in\mathds{N}}
    \big| f_n(x^2) \big| \Big)
\]
\end{document}
```

2. Compile, and take a look at the formula:

$$\lim_{n\to\infty} \sup_{x\in\mathbb{R}} f_n(x^2) < n\Big(\sum_{x\in\mathbb{R}, n\in\mathbb{N}} \big| f_n(x^2) \big| \Big)$$

Although the LaTeX code is fine, you will notice that several points should be improved:

- The subscripts below the operators on the left-hand side of the equation are vertically misaligned
- The wide space around the sum symbol doesn't look good
- The exponent 2 is a bit higher than the parentheses

We will fix this now.

How to do it...

The `mathtools` package offers a lot of improvements and tools for writing math. It is a recommended companion to the `amsmath` package. Here's how you can use it:

1. Add the command `\usepackage{mathtools}` to the preamble.
2. Write the `\adjustlimits` command right before the `\lim` operator.
3. Insert `\smashoperator{` before the `\sum` and write the argument closing brace `}` after the subscript.
4. Change `x^2` to `\cramped{x^2}` to slightly lower the exponent, at both places. Now, your formula reads like this:

```
\[
  \adjustlimits\lim_{n\to\infty} \sup_{x\in\mathbb{R}}
    f_n(\cramped{x^2})
    < n \Big( \smashoperator{\sum_{x\in\mathbb{R},
    n\in\mathbb{N}}}
    \big\lvert f_n(\cramped{x^2}) \big\rvert \Big)
\]
```

5. Compile, take a look, and compare the output with that of with the original formula:

$$\lim_{n\to\infty} \sup_{x\in\mathbb{R}} f_n(x^2) < n\Big(\sum_{x\in\mathbb{R},n\in\mathbb{N}} \big| f_n(x^2) \big| \Big)$$

How it works...

When you just load the `mathtools` package, it implicitly loads the `amsmath` package as well. It fixes known `amsmath` errors. The `amsmath` package is very static, which is one reason why the `mathtools` package has been created. It provides further tools that have been collected over years, written by various authors for tweaking the math layout.

Some of the tools we have just used, such as:

- `\adjustlimits`: This command aligns the limits of two consecutive operators. They are actually arguments. While you can simply write it right in front of the two operators, it has this exact syntax:

  ```
  \adjustlimits{operator1}_{limit1} {operator2}_{limit2}
  ```

- `\smashoperator`: This command ignores the width of subscript and superscript.
- `\cramped`: This command forces a compact TeX style. This way, exponents will placed a bit lower than default. This is especially useful for inline math, to avoid undesired stretching of line spacing.

Automatic line-breaking in equations

Usually, we pay particular attention to formula design. In case of multiline formulas, we manually choose the best break point and where to align things. But imagine a long chain of calculations, such as in proofs or in math homework. It would be great if TeX could wrap displayed formulas as it does in case of normal text. And it's possible.

How to do it...

The `breqn` package is designed for exactly this purpose. This recipe will demonstrate it. We will use the `beamer` class because seminar slides are usually limited in terms of space. Follow these steps:

1. Specify the document class:

   ```
   \documentclass[12pt]{beamer}
   ```

2. As the `beamer` class uses sans-serif math font by default, we switch to the serif math font as in standard documents:

   ```
   \usefonttheme[onlymath]{serif}
   ```

3. Load the `breqn` package:

   ```
   \usepackage{breqn}
   ```

4. Start the document, and begin a frame:

   ```
   \begin{document}
   \begin{frame}
   ```

5. Write your math formula, without paragraph breaks and without space, just as you would do with simple math. You can use line breaks in the editor, but you don't need to. The important point is: use a `dmath*` environment.

   ```
   \begin{dmath*}
     \left( \frac{f}{g} \right)^\prime(x)
       = \lim_{h \rightarrow 0}
         \left( \frac{1}{ g(x+h) g(x) } \right)
         \left[
                \frac{ f(x+h) - f(x) }{h} g(x)
               -\frac{ g(x+h) - g(x) }{h} f(x)
         \right]
       = \frac{f^\prime(x)g(x)-f(x)g^\prime(x)}{g^2(x)}
   \end{dmath*}
   ```

6. End the frame and the document:

```
\end{frame}
\end{document}
```

7. Compile, and take a look:

$$\left(\frac{f}{g}\right)'(x) = \lim_{h\to 0}\left(\frac{1}{g(x+h)g(x)}\right)\left[\frac{f(x+h)-f(x)}{h}g(x) - \frac{g(x+h)-g(x)}{h}f(x)\right]$$
$$= \frac{f'(x)g(x)-f(x)g'(x)}{g^2(x)}$$

How it works....

The primary feature of the `breqn` package is dealing with overlong displayed formulas. When a formula is too wide, it will be broken automatically with a suitable break point. The following lines will be indented. Usually, they will start with a relation symbol.

A great implicit feature is supporting automatically sized delimiters across line breaks. This means the commands `\left` and `\right` together with a delimiter such as a parenthesis. Classic multiline math environments, such as `align` and `gather`, would bring an error if there's one on a line but the counterpart on another line. Usual fixes are inserting the `\right.` and `\left.` commands, which produce an invisible delimiter, but then it's not guaranteed that the measured height of both lines would be equal. So you could end up with delimiter pairs of different size or doing it manually by using the `\big`, `\Big`, `\bigg`, or `\Bigg` operators. The `breqn` package fixes this for you.

You could give the `breqn` package a try, even if it's for specific cases only, such as the `\left` and `\right` issues.

Besides that, it can really speed up writings such as math homework and proofs with a lot of longish calculations.



The package authors recommended to load `breqn` after all other math related packages, such as `amsmath`, `amssymb`, `mathpazo`, or `mathptmx`. So, for example, `breqn` detects and uses options provided to amsmath, such as `leqn` and `fleqn`. A good rule of thumb is: load sophisticated or late developed packages late. But in case of problems, its worth a try to change loading order.

Highlighting in a formula

In a complex formula or equation, it can be useful to emphasize a specific part. We can achieve this, for example, through highlighting by color or by framing. This is especially useful in presentations, where we could change the highlighted area from frame to frame while explaining.

We will take the challenge to work on more than simple basic math. So we can verify that the method works in arbitrary situations. So our original material will be the `amsmath` matrices. We will illustrate a matrix transposition by highlighting submatrices and drawing arrows.

How to do it...

We will use TikZ for this. Such a big graphics package may seem to be a bit heavy for such a purpose at first. But it provides us with consistent styles and allows us to do nearly anything graphically.

Let's do an example step by step:

1. Start with a document class:
```
\documentclass{article}
```
2. Load the `amsmath` package, so we can write matrices:
```
\usepackage{amsmath}
```
3. Load the `tikz` package:
```
\usepackage{tikz}
```
4. In addition, load the `fit` library, which we will use for auto-fitting to nodes:
```
\usetikzlibrary{fit}
```
5. Define a macro for adding overlays:
```
\newcommand{\overlay}[2][]{\tikz[overlay,
  remember picture, #1]{#2}}
```
6. Declare a style that shall be applied for a highlighted area:
```
\tikzset{
  highlighted/.style = { draw, thick, rectangle,
                         rounded corners, inner sep = 0pt,
                         fill = red!15, fill opacity = 0.5
                       }
}
```

7. Define a command for highlighting an area. The area shall be rectangular and fit around nodes with the names `left` and `right`. The preceding style will be applied. A name as argument will be stored as the name for this area for later referencing.

```
\newcommand{\highlight}[1]{%
  \overlay{
    \node [fit = (left.north west) (right.south east),
           highlighted] (#1) {}; }
}
```

8. Define another command that prints text as a node and flags it with a name:

```
\newcommand{\flag}[2]{\overlay[baseline=(#1.base)]
  {\node (#1) {$#2$};}}
```

9. After those preparations, begin with your document:

```
\begin{document}
```

10. Open an equation:

```
\[
```

11. Write a matrix using the `pmatrix` environment of the `amsmath` package. In any cell which shall be flagged for later use, use the command `\flag{name}{content}` for the content.

```
M = \begin{pmatrix}
  \flag{left}{1} & 2 & 3 & 4 & 5 \\
  6 & 7 & 8 & 9 & 10 \\
  11 & \flag{before}{12} & \flag{right}{13} & 14 & 15 \\
  16 & 17 & 18 & 19 & 20
\end{pmatrix}
```

12. Add the first highlighting to a sub matrix and give it the name `N`:

```
\highlight{N}
```

13. Add some horizontal space. Write the transposed matrix with desired flags. Again, highlight the interesting part, which is the transposed submatrix. We choose the name `NT`. Then end the equation.

```
\qquad
M^T = \begin{pmatrix}
  \flag{left}{1} & 6 & 11 & 16 \\
  2 & 7 & \flag{after}{12} & 17 \\
  3 & 8 & \flag{right}{13} & 18 \\
```

```
    4 & 9 & 14 & 19 \\
    5 & 10 & 15 & 20
  \end{pmatrix}
  \highlight{NT}
\]
```

14. Now we will add further bells and whistles. Start our `\overlay` macro:

```
\overlay{
```

15. Draw a thick, red, dotted line between interesting nodes, which we flagged previously using the names `before` and `after`:

```
  \draw[->, thick, red, dotted] (before) -- (after);
```

16. Draw another such line, this time dashed, between our submatrices:

```
  \draw[->, thick, red, dashed] (N) -- (NT)
   node [pos=0.68, above] {Transpose};
```

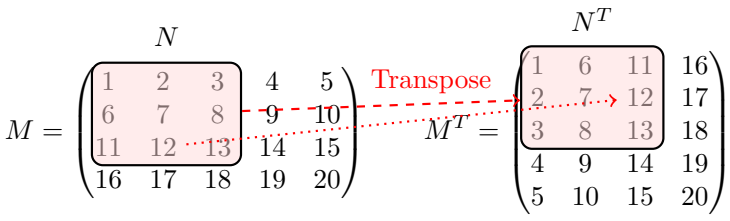
17. Add labels to the sub matrices, and then add the closing brace for the `\overlay` command:

```
  \node[above of = N ] { $N$   };
  \node[above of = NT] { $N^T$ };
}
```

18. End the document:

```
\end{document}
```

19. Compile the document twice. After the second run, our result is as follows:



How it works...

The key is the small `\overlay` macro, which is used this way:

```
\overlay[options]{drawing commands}
```

It creates a TikZ picture with three properties:

- The `overlay` option ensures that the bounding box of the current picture remains unchanged. It's like drawing but without requiring space.
- The `remember picture` option lets TikZ remember the position of the current picture. It writes that value into the `.aux` file. Using the value from the `.aux` file is the reason why a second run may be required for correct positioning.
- The third property is determined by the optional argument of the `\overlay` command, which is denoted by `#1` and is empty by default. This allows adding options.

Our next key command is the `\flag` macro:

```
\flag{name}{content}
```

It prints the content while flagging it with a certain name. Internally, it calls the `\overlay` macro. It adds the option `baseline=(#1.base)`, whose achievement is that the baseline of the generated node will be the baseline of the content, not the center.

We come to the highlighting. At first, we used the `\tikzset` command to declare a global style with the name `highlighted`. We chose the form of a rounded rectangle and added color and transparency. You could freely customize this style using colors, shapes, shadings, and more, whatever TikZ provides.

The command `\highlight{name}` again uses the `\overlay` macro to add our style to an area. It requires that we already flag two positions using the names `left` and `right`. It uses TikZ's fit feature to calculate a rectangular area which exactly fits to the given positions. In our case, these are the top-left corner (north-west of the left node) and the bottom-right corner (south-east of the right node).

After we added any flags we wanted, we created another overlay with some drawing commands. We can use any TikZ drawing command together with the names we defined using the `\flag` command. So we added arrows and labels to the drawing.

We did a lot at once in this recipe. We colored areas, and we added labels and arrows, relatively positioned to the content. Once we have those sophisticated small macros in our document, we can add such flags in text, math, lists, tables, drawings, and anywhere we like. This allows us to freely draw over our document, independent of the underlying layout.

There's more...

The `tikzmark` package has basically the same approach and can be used in a similar way. We took the direct way for maximum control though you may consider using that package.

If you don't need something very fancy, there are basic commands such as `\textcolor` for coloring and `\boxed` in the `amsmath` package. The `empheq` package is another option for emphasizing equations or parts of them. As usual, you can check out their documentation by typing the `texdoc` command in Command Prompt or by visiting `http://texdoc.net`.

The same applies to TikZ, of course; type the `texdoc tikz` command in Command Prompt or visit `http://texdoc.net/pkg/tikz`. The web site `http://texample.net`, maintained by me, contains similar examples like this recipe and hundreds more of graphical examples.

Stating definitions and theorems

Strictly structured math documents often contain certain numbered textual elements, which can be definitions, theorems, lemmas, examples remarks, and so on. Numbering is mainly for cross-referencing. Such structuring is especially the case in self-contained scientific works such as theses.

As a sample application, we will create a definition, a theorem with a proof, a lemma, and an additional note. All shall be automatically numbered for cross-referencing. Just for fun, we will apply it to school geometry: the theorem of Pythagoras.



Actually, this recipe is of pretty universal use: you can create numbered, named environments, ready for cross-referencing, whatever it shall contain at the end.

How to do it...

We continue using the `amsmath` package, with a special part of it, the `amsthm` package as follows:

1. As usual, start with a document class. For our recipe, the `article` class is sufficient.

   ```
   \documentclass{article}
   ```

2. Load the packages `amsmath` and `amsthm`:

   ```
   \usepackage{amsmath}
   \usepackage{amsthm}
   ```

3. Define a theorem environment with the internal name `thm`, printing `Theorem` in the document:

```
\newtheorem{thm}{Theorem}
```

4. Do the same for the `lem` package as `Lemma` environment:

```
\newtheorem{lem}{Lemma}
```

5. Switch the theorem style to `definition`:

```
\theoremstyle{definition}
```

6. Define a definition environment with the internal name `dfn`, printing `Definition`:

```
\newtheorem{dfn}{Definition}
```

7. Switch the theorem style to `remark`:

```
\theoremstyle{remark}
```

8. Define an unnumbered environment for notes by inserting a star:

```
\newtheorem*{note}{Note}
```

9. Start the document:

```
\begin{document}
```

10. Write up a definition:

```
\begin{dfn}
  The longest side of a triangle with a right angle
  is called the \emph{hypotenuse}.
\end{dfn}
```

11. Add a note:

```
\begin{note}
  The other sides are called \emph{catheti},
  or \emph{legs}.
\end{note}
```

12. Write a theorem. Give it the optional name `Pythagoras`. Insert a label for cross-referencing:

```
\begin{thm}[Pythagoras]
  \label{pythagoras}
  In any right triangle, the square of the hypotenuse
  equals the sum of the squares of the other sides.
\end{thm}
```

13. Continue with the proof. We keep it short here. Let's have a cross-reference to a later written lemma:

```
\begin{proof}
  The proof has been given in Euclid's Elements,
  Book 1, Proposition 47. Refer to it for details.
  The converse is also true, see lemma \ref{converse}.
\end{proof}
```

14. Add a lemma with an internal label:

```
\begin{lem}
  \label{converse}
  For any three positive numbers \(x\), \(y\),
  and \(z\) with \(x^2 + y^2 = z^2\), there is a
  triangle with side lengths \(x\), \(y\) and \(z\).
  Such triangle has a right angle, and the hypotenuse
  has the length \(z\).
\end{lem}
```

15. Add another note. Refer to the theorem of Pythagoras:

```
\begin{note}
  This is the converse of theorem \ref{pythagoras}.
\end{note}
```

16. End the document:

```
\end{document}
```

17. Compile twice to get the cross-references right. Examine how our environments are printed:

Definition 1. The longest side of a triangle with a right angle is called the *hypotenuse*.

Note. The other sides are called *catheti*, or *legs*.

Theorem 1 (Pythagoras)**.** *In any right triangle, the square of the hypotenuse equals the sum of the squares of the other sides.*

Proof. The proof has been given in Euclid's Elements, Book 1, Proposition 47. Refer to it for details. The converse is also true, see lemma 1. □

Lemma 1. *For any three positive numbers x, y, and z with $x^2 + y^2 = z^2$, there is a triangle with side lengths x, y and z. Such triangle has a right angle, and the hypotenuse has the length z.*

Note. This is the converse of theorem 1.

How it works...

The `amsthm` package provides the `\newtheorem` command, which creates an environment for us. Its special property is that it has a counter which is automatically increased, usable for cross-referencing if we would provide a label. Furthermore, notation and body text get a formatting as you can see in the output:

- The default style is `plain`. The label is printed in bold, the optional name in parenthesis in normal upright font, and the body text in italics.
- The `definition` style generates a bold label and normal upright body text.
- The `note` style produces an italic label and normal upright body text.

When you switch the style by using the `\theoremstyle` command, it becomes valid for the following `\newtheorem` definitions.

You can choose a suitable name for the environment. Note, that the name should not be already used by TeX or LaTeX. So, the `\newtheorem{def}{Definition}` command would not work as `\def` is already a command.

We can insert cross-references as usual: insert a `\label` command with a name. Then you can use the `\ref` command anywhere to point to the number of the environment where we placed the label. Usually, environments that are not referenced, such as notes or corollaries, can be defined without numbers if you use the starred form the `\newtheorem*` command, like we did previously for notes.

The `proof` environment is a special predefined environment, which starts with `Proof` and automatically gets an endmark, the *quod erat demonstrandum* symbol.

The `amsthm` package provides a command, `\newtheoremstyle`, which takes nine arguments, so you can customize fonts and spacing of the environment, also specifically for head and body. You can find the parameters in the manual, accessible by typing the `texdoc amsthm` command in Command Prompt or online at `http://texdoc.net/pkg/amsthm`. So you can add your own designs.

There's more...

While we refer to the manual for styling the design, numbering is of fundamental importance.

Adjusting the numbering

If you have a lot of definitions, theorems, lemmas, and such, you may decide not to number all these things separately. That's because it feels a bit strange if theorem 5 is followed by lemma 2. Counting all together makes look-up easier too. There's an option to use an already defined counter, give it in square brackets after the name. Use the following for our recipe:

```
\newtheorem{thm}{Theorem}
\newtheorem{lem}[thm]{Lemma}
\newtheorem{dfn}[thm]{Definition}
```

Now, one counter is valid for theorems, lemmas, and definitions. That's the `thm` counter.

There's another option for numbering per sectional unit, such as per section or per chapter. For example, in order to count per chapter when the class supports chapter, write this:

```
\newtheorem{thm}{Theorem}[chapter]
```

You can do it per section as follows:

```
\newtheorem{thm}{Theorem}[section]
```

If you would like to begin with a number, (say, to have "1.5 Theorem" instead of "Theorem 1.5"), insert the `\swapnumbers` command before the theorem definitions in your preamble.

An alternative theorem package

The `ntheorem` package is a good alternative to the `amsthm` package. Basically, it provides compatible styles, so the result will be similar if you load it instead of the `amsthm` package in this way:

```
\usepackage[amsmath,amsthm,thmmarks]{ntheorem}
```

An interesting property is the better placement of endmarks. When a proof ends with a displayed equation, the `amsthm` package sets the endmark not next to it but shifted a bit below. The `ntheorem` package doesn't have this issue. However, I would never end a proof with a formula or diagram at the end without additional work, just as I would not be silent in a seminar presentation after showing a final formula. A final word is always good, so I don't consider it as an issue.

Besides that, the `ntheorem` package brings its own styles and a different manner of customization. As expected, you can take a look at the package documentation by typing the `texdoc ntheorem` command in Command Prompt or by visiting `http://texdoc.net/pkg/ntheorem` to read the package reference.

Additional theorem tools

There's another package which adds great value, no matter whether you use the `amsthm` or `ntheorem` package. It's called `thmtools` and here is what it provides:

- An automatically generated list of theorems, similar to lists of tables and figures
- The ability to fully repeat a theorem statement
- A `key = value` interface for numbering style and further properties
- Support for intelligent referencing, such as automatically adding the theorem name to a reference
- Shaded and boxed designs (implicitly using the `shadethm` and `thmbox` packages)
- Even more options for tweaking design

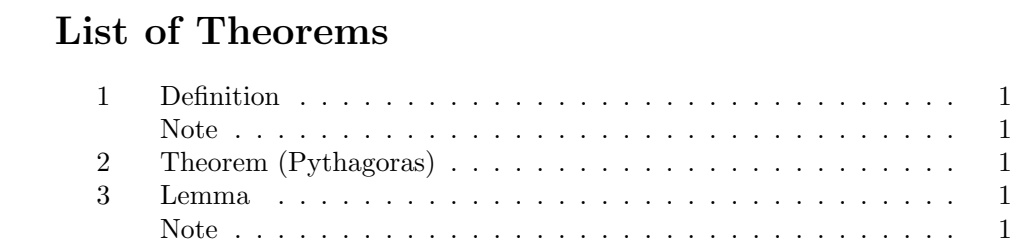
So, for a list of theorems, simply add this to your preamble:

```
\usepackage{thmtools}
```

Later in your document, call the following command:

```
\listoftheorems
```

Compile at least twice. For our small recipe document with shared numbering, it results in this short list:



You see, giving names to definitions and lemmas, like we did with the theorem, adds some value.

Finally, let's take a look at emphasizing by shading and boxing using the `thmtools` package. You can apply them this way:

1. Load the package `thmtools`:

   ```
   \usepackage{thmtools}
   ```

2. Load the package `xcolor` so you can use color definitions:

   ```
   \usepackage{xcolor}
   ```

3. Instead of using the `\newtheorem` command, use the following command with a color you like:

   ```
   \declaretheorem[shaded={bgcolor=red!15}]{Theorem}
   ```

4. For definitions, take a box design instead of the `\newtheorem` command and use the following command with a color you like:

   ```
   \declaretheorem[thmbox=L]{Definition}
   ```

5. In the body text, write your definitions as follows:

   ```
   \begin{Definition}[name]
     ...
   \end{Definition}
   ```

6. Write your theorems like this:

   ```
   \begin{Theorem}[name]
     ...
   \end{ Theorem}
   ```

Applied to our recipe, we get the following result:

Definition 1

The longest side of a triangle with a right angle is called the hypotenuse.

Theorem 1 (Pythagoras)**.** *In any right triangle, the square of the hypotenuse equals the sum of the squares of the other sides.*

To further explore the package, take a look at the manual by typing the `texdoc thmtools` command in Command Prompt or by visiting `http://texdoc.net/pkg/thmtools`.

Drawing commutative diagrams

In algebra, especially in category theory, we use so-called **commutative diagrams**. Vertices denote objects such as groups or modules. Arrows represent morphisms, which are maps between those objects. The characteristic quality of such diagrams is that they commute. This means that you get the same result by composition, no matter which directed way in the diagram you go, as long as the start point and end point are the same.

Such diagrams are used a lot for visualizing algebraic properties. Whole proofs are done by chasing through such a diagram. That's why our next recipe will deal with it. We will start with a diagram for the first isomorphism theorem in group theory.

How to do it...

We will use the TikZ package. While we use just a few features for now, it offers a wealth of arrow heads and tails and useful graphic tools for positioning and labeling. Here are the steps you need to follow:

1. Start with a document class:
   ```
   \documentclassarticle}
   ```
2. Load the `tikz` package:
   ```
   \usepackage{tikz}
   ```
3. Load the `matrix` library of TikZ:
   ```
   \usetikzlibrary{matrix}
   ```
4. Load the `amsmath` package:
   ```
   \usepackage{amsmath}
   ```
5. Declare any operators you will need that are not defined yet:
   ```
   \DeclareMathOperator{\im}{im}
   ```
6. Start the document and begin a TikZ picture environment:
   ```
   \begin{document}
   \begin{tikzpicture}
   ```
7. Use the `\matrix` command. First, provide a name to it. We choose `m`.
   ```
   \matrix (m)
   ```

8. Provide options for the matrix. We declare that all nodes shall have content in math mode. Furthermore, we define the row and column distance.

```
[
  matrix of math nodes,
  row sep    = 3em,
  column sep = 4em
]
```

9. The matrix content follows in curly braces, followed by a semicolon. It's just like in a normal `tabular`, `array`, or `matrix` environment. But also note that the last line needs a line break with the `\\` symbol at the end.

```
{
  G              & \im \varphi \\
  G/\ker \varphi &             \\
};
```

10. Now we will draw the arrows. Each will be done as an edge with an arrow tip. Begin with a path and add the desired arrow tip as an option:

```
\path[-stealth]
```

11. Draw an edge. It goes between matrix nodes, which can be addressed using the implicitly give name (*m-row-column*). Following an edge, we add a node that is used as label.

```
(m-1-1) edge node [left] {$\pi$} (m-2-1)
```

12. Draw the next edge. This time, we will use the `|-` operator to definitely get a horizontal edge:

```
(m-1-1.east |- m-1-2)
  edge node [above] {$\varphi$} (m-1-2)
```

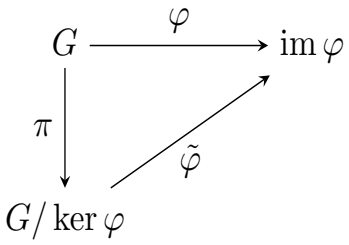
13. Draw the last edge and end the path with a semicolon:

```
(m-2-1) edge [dashed] node [below] {$\tilde{\varphi}$}
  (m-1-2);
```

14. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

15. Compile, take a look at the diagram:



How it works....

Besides loading the packages, the basic procedure is this:

1. Define macros, operators, or styles.
2. Put all objects in a matrix.
3. Draw labeled arrows as edges with nodes.

As part of step 1, we just defined the operator `im` for the "image" of a map. The operator `ker` is already defined.

In step 2, we chose a matrix of math nodes, so all node content is in math mode. While we just define the spacing in rows and columns, you could define even styles for specific rows or columns. Refer to the TikZ manual for further possibilities.

In step 3, we made every edge an arrow. This is inherited from the path option `-stealth`, which produced an arrow with a stealth tip.

We could apply tip and tail styles for single arrows as well. Let's have a quick demonstration. Load the `arrows.meta` library:

```
\usetikzlibrary{arrows.meta}
```

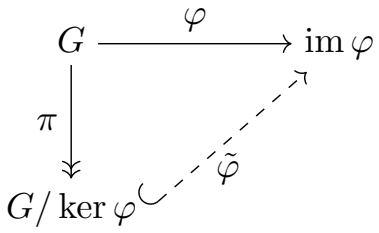


You can load several libraries at once by using the `\usetikzlibrary{matrix,arrows.meta}` command.

Now you have a lot of customizable styles for arrow tips, which you can use at both ends. There are barbs, harpoons, brackets, caps, triangles, and many more than just typical arrow tip styles. And you can customize them via options. The TikZ manual provides an arrow tips reference section. Here, we have the space for a short sample. After you load the `arrows.meta` library, change the path as follows:

```
\path
  (m-1-1) edge [->>] node [left] {$\pi$} (m-2-1)
  (m-1-1.east |- m-1-2)
    edge [->] node [above] {$\varphi$} (m-1-2)
  (m-2-1.east) edge [{Hooks[right,length=0.8ex]}->,
    dashed] node [below] {$\tilde{\varphi}$} (m-1-2);
```

The diagram now looks like this:

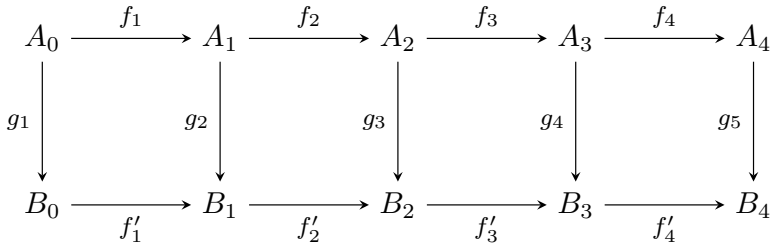


We gave the arrow specification as an option to each edge. While the dash `(-)` means the base edge, tail and head come before or after. So, the `->>` symbol has a double head (commonly used for subjective maps), while the `[{Hooks[right,length=0.8ex]}->` command starts with a hook and ends with a single arrow head, which stands for an injective map. Furthermore, the edge is dashed, often used to mark the unique, structure-preserving map that makes the diagram commute.

All this sounds all like a very elaborate science, but it's worth the effort as having this tuning power is better than having no choice.

There's more...

Diagrams can become complicated. They can have more columns, more rows, and many more arrows and labels. In such cases, `for` loops and calculation options can be a relief. Look at this diagram:



It's used to prove the so-called five lemma. Let's see how to generate it efficiently:

1. Start the document similar to our previous example. In addition, load the `calc` library:

```
\documentclass{article}
\usepackage{tikz}
\usetikzlibrary{matrix,calc}
\begin{document}
```

2. Start the picture. This time, we use the arrow options for the whole picture and define a label style to have labels in smaller size:

```
\begin{tikzpicture}[-stealth,
  label/.style = { font=\footnotesize }]
```

3. Write the matrix:

```
\matrix (m) [
    matrix of math nodes,
    row sep    = 4em,
    column sep = 4em ]
{ A_0 & A_1 & A_2 & A_3 & A_4 \\
   B_0 & B_1 & B_2 & B_3 & B_4 \\ };
```

4. Now let's save our work using a `for` loop:

```
\foreach \i in {1,...,4} {
\path
    let \n1 = { int(\i+1) } in
      (m-1-\i) edge node [above, label] {$f_\i$}
        (m-1-\n1)
      (m-2-\i) edge node [below, label] {$f^\prime_\i$}
        (m-2-\n1)
      (m-1-\i) edge node [left, label] {$g_\i$} (m-2-\i);
}
```

5. One arrow to go, then we end the picture and the document:

```
  \path (m-1-5) edge node [left, label] {$g_5$} (m-2-5);
\end{tikzpicture}
\end{document}
```

That's it!

The `\foreach` command repeats the action for the value of `1`, `2`, `3`, `4`, which is taken as loop variable `\i` here.

For each path, we calculated `\n1` as `\i+1`, since we draw edges to the next node. We use the `\let` command, provided by the `calc` library. Roughly said, we can use it for numbers like so:

```
\path let \n1 = { formula } in … <drawing using
  number n1 somewhere>
```

We can also use it for points like this:

```
\path let \p1 = { formula } in … <drawing using the
  point p1 somewhere>
```

You can have several expressions in a `let` command. The `let` command is described in depth in the TikZ library, but you probably already got the meaning.

Plotting functions in two dimensions

Functions play an important role in mathematics. To visualize them to better understand their properties, so that we can see roots and extreme points, we can plot them in a coordinate system. In this recipe, we will see how to easily plot functions. First, let's print the polynomial function *f(x) = x^3 - 5*x*.

How to do it...

We will use the `pgfplots` package, which is based on PGF/TikZ. We already used it in the previous chapter to draw various diagrams. Now it will plot a function for us. Follow these steps:

1. Start with a document class. For this recipe, we decide on the `standalone` class, which is great for creating single graphics with just a small margin, but you can choose the `article` class or another one as well.

```
\documentclass[border=10pt]{standalone}
```

2. Load the `pgfplots` package:

```
\usepackage{pgfplots}
```

3. Start the document and open a `tikzpicture` environment:

```
\begin{document}
  \begin{tikzpicture}
```

4. Begin an `axis` environment with centered axis lines:

```
    \begin{axis} [axis lines=center]
```

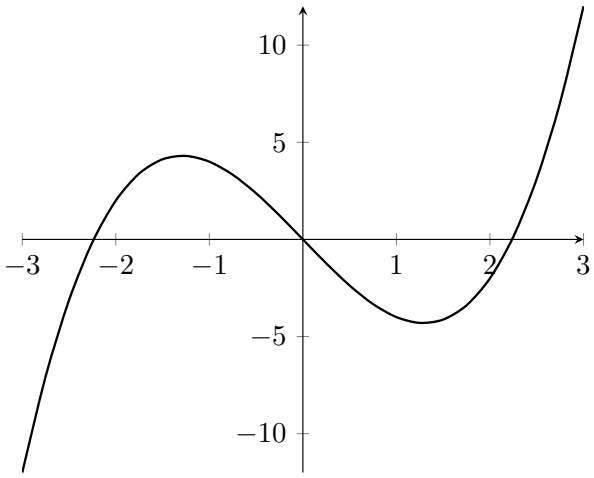
5. Call the plot command. As an option, use the domain or the range for *x*. Choose a thick line and smooth plotting:

```
      \addplot [domain=-3:3, thick, smooth] { x^3 - 5*x };
```

6. End the `axis`, `tikzpicture`, and `document` environments:

```
    \end{axis}
  \end{tikzpicture}
\end{document}
```

7. Compile, and take a look:



How it works...

Within a `tikzpicture` environment, we placed an `axis` environment, provided by the `pgfplots` package. The `axis` environment takes all options for lines, ticks, labels, grids, and style that apply to the coordinate system and the whole plot appearance. We will see some of them later on and also in the next recipe.

The `\addplot` command gets the function formula. It understands options specifically for the single plot, such as domain, number of samples, color, thickness, and other styles. You can have several `\addplot` commands within a single `axis` environment to get several plots in one drawing.

Our first function plot required a formal setup using environments, but we effectively got the plot using a few simple commands.

There's more...

Now that we have the basic steps for plotting, we will work out some different plotting styles.

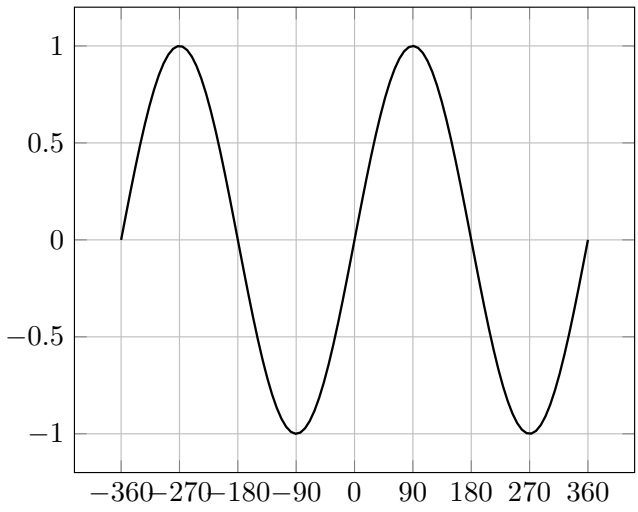
Adding ticks and grids

At the start of our recipe, we used classical centered axis lines. By default, though, an axis will be a rectangular box that contains the plot. Let's look at this. In addition, we will add a grid and choose the places for the axis ticks with labels.

The settings of our plots are always the same, within a `tikzpicture` environment, so we can reduce the code shown here to the relevant `axis` environment:

```
\begin{axis} [grid, xtick = {-360,-270,...,360}]
  \addplot [domain=-360:360, samples=100, thick] { sin(x) };
\end{axis}
```

This gives us the following result:

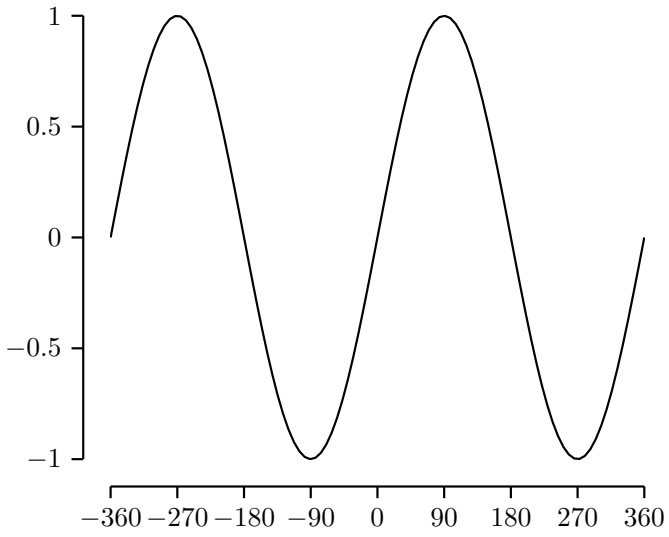


Reducing axes

While a grid and a lot of ticks can be useful to inspect specific values of a function, an overall view into a function can be improved with reduced axes, maybe even shifted away a bit. There's a style which you can use. If you download the file at `http://pgfplots.net/media/tikzlibrarypgfplots.shift.code.tex` and put it into your document folder. You can load this style in your preamble with the following command:

```
\usepgfplotslibrary{shift}
```

Now, just change the `axis` option `grid` to `shift` and you will get this:



While the default shift value is 10 pt, you can adjust it, such as by changing the code to `shift=15pt`.

The same library can be used in three dimensions, and we will return to this in our next recipe.

Plotting in polar coordinates

The standard `axis` environment is used for Cartesian coordinates. However, the `pgfplots` package also provides logarithmic axes and polar axes.

Functions can also be defined in polar coordinates. In a polar coordinate system, each point is determined by the distance from the origin `0` and by the angle to a reference axis. Now the argument of a function is expected to be an angle, and the function value is considered as the distance from the origin. Let's try a polar plot:

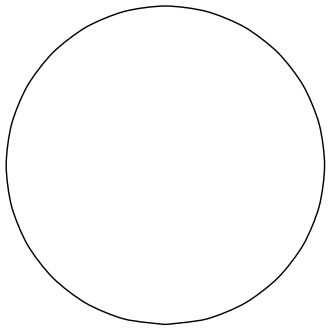
1. In your document preamble, add this after the `\usepackage{pgfplots}` command:

   ```
   \usepgfplotslibrary{polar}
   ```

2. Within a `tikzpicture` environment, like we just saw, write the axis with the plot using the `polaraxis` package, just hiding the axis lines this time:

```
\begin{polaraxis}[hide axis]
  \addplot[domain=0:180,smooth] {sin(x)};
\end{polaraxis}
```

3. Compile, and you will get a simple circle:

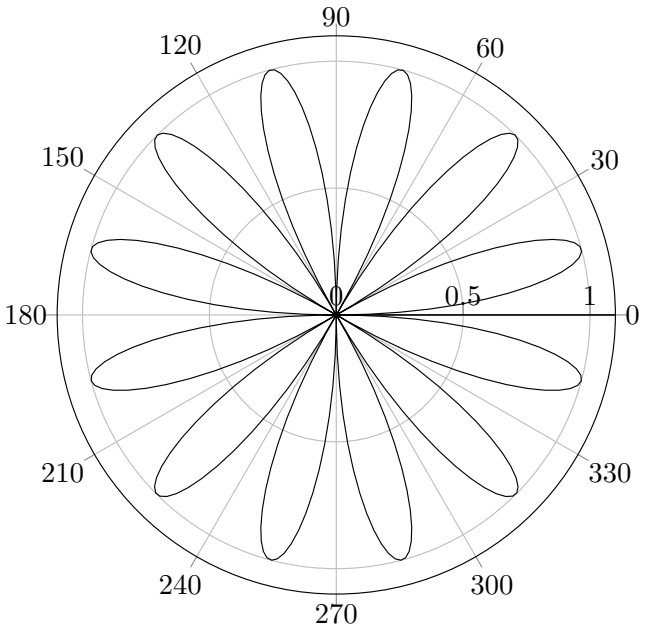


So the sine function, which was a wave in our preceding Cartesian coordinate plot, now looks like a circle in polar coordinates, demonstrating that changing the axis can give useful visual insights.

A complete polar plot shows angles, a radius, and a grid with circular and radial lines. The preceding sine circle gave a circle for 180 degrees. So if we got, let's say, a factor of 6 in the argument, we should get a squished circle every 30 degrees:

```
\documentclass[border=10pt]{standalone}
\usepackage{pgfplots}
\usepgfplotslibrary{polar}
\begin{document}
\begin{tikzpicture}
  \begin{polaraxis}
    \addplot[domain=0:360,samples=300] {sin(6*x)};
  \end{polaraxis}
\end{tikzpicture}
\end{document}
```

Compiling this document results in this output:



Plotting in three dimensions

Functions with two arguments can be visualized in three-dimensional plots. Getting this on paper or into a PDF is a bit more challenging. We need a projection, a view point or angle, and a depth, which means that some parts will be hidden and others in the front.

At the end of the previous recipe, we plotted the function *f(x) = sin(x)*. Now, when we add a dimension, we will plot *f(x,y) = sin(x)*sin(y)*.

How to do it...

Like in the previous recipe, we will use the `pgfplots` package. Follow these steps:

1. Start with a document class. Like in the previous recipe, we use the `standalone` class. But it's fine if you choose the `article` class instead.

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `pgfplots` package:

   ```
   \usepackage{pgfplots}
   ```

3. Start the document, and open a `tikzpicture` environment:

```
\begin{document}
  \begin{tikzpicture}
```

4. Begin an `axis` environment with options:

```
    \begin{axis} [
      title = {$f(x,y) = \sin(x)\sin(y)$},
      xtick = {0,90,...,360},
      ytick = {90,180,...,360},
      xlabel = $x$, ylabel = $y$,
      ticklabel style = {font = \scriptsize},
      grid
    ]
```

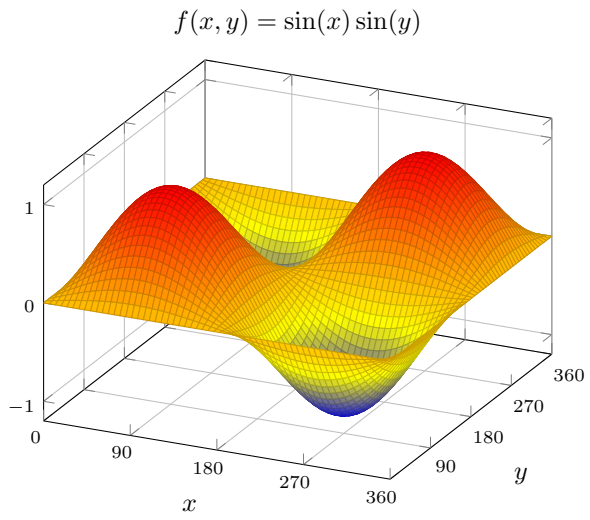
5. Call the 3D plot command. Use the surface style, specify the domain (that is, the range for *x* and *y*), and choose a number of samples:

```
      \addplot3 [surf, domain=0:360, samples=60]
    { sin(x)*sin(y) };
```

6. End the `axis`, `tikzpicture`, and `document` environments:

```
    \end{axis}
  \end{tikzpicture}
\end{document}
```

7. Compile, and take a look:



How it works...

The basics of `pgfplots` and the `axis` environment were already explained in the previous recipe.

This time, we used the `\addplot3` command in analogy to the `\addplot` command in the previous recipe, but now with two variables.



The 3 in the name of `\addplot3` stands for 3-dimensional. As `\addplot` already is the two-dimensional version, there's not `\addplot2` command.

The `\addplot3` command understands similar options. In addition, you can choose a 3D-specific style. We chose the `surf` operator for a surface plot.

There's more...

We will take a look at another axis style.

Reducing axes

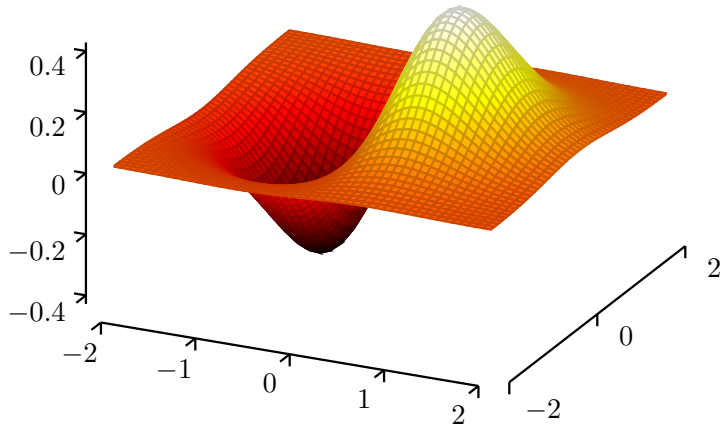
As promised in the previous recipe, we will apply a style for reduced and shifted axes. If you don't have it already, download the file at `http://pgfplots.net/media/tikzlibrarypgfplots.shift.code.tex` and put it into your document folder. Load this style in your preamble with this command:

```
\usepgfplotslibrary{shift}
```

Now, we can use the `shift` style. To see a new function, we will use it with an exponential function. The brief but complete code shall be as follows:

```
\documentclass[border=10pt]{standalone}
\usepackage{pgfplots}
\usepgfplotslibrary{shift}
\begin{document}
\begin{tikzpicture}
  \begin{axis} [shift3d]
    \addplot3 [surf, colormap/hot2, domain = -2:2, samples = 50]
      { x/exp(x^2+y^2) };
  \end{axis}
\end{tikzpicture}
\end{document}
```

Compile, and see the distraction-free plot:



The default shift value is 10 pt but you can adjust it, say by using `shift3d=15pt`.

In the previous recipe the shifted axis style was used in two dimensions. Here, it is for three dimensions.

Drawing geometry pictures

A well-known classic field of mathematics is geometry. You may know Euclidean geometry from school, with constructions involving compass and ruler. Math teachers may be very interested in drawing geometric constructions and explanations. Underlying constructions can help us with general drawings where we would need intersections and tangents of lines and circles, even if it does not look like geometry.

So, in this recipe we will remember school geometry drawings.

How to do it...

We will use the `tkz-euclide` package, which works on top of TikZ. As our first goal, we will construct an equilateral triangle. Then we will add some information. Follow these steps:

1. Start with a document class. It could be any one; here we can use the `standalone` class to focus on a single image.

```
\documentclass[border=10pt]{standalone}
```

2. Load the `tkz-euclide` package:

```
\usepackage{tkz-euclide}
```

3. Specify which geometric objects shall be supported. Let's take all available:

```
\usetkzobj{all}
```

4. Start the document and open a TikZ picture environment:

```
\begin{document}
\begin{tikzpicture}
```

5. Define some starting points:

```
\tkzDefPoint(0,0){A}
\tkzDefPoint(4,1){B}
```

6. Calculate further points. Here we will take an intersection of the circle around A through B with the circle around B through A:

```
\tkzInterCC(A,B)(B,A)
```

7. Get the calculated points and give them the names C and D:

```
\tkzGetPoints{C}{D}
```

8. Now we start drawing. Draw the triangle with the corners A, B, and C:

```
\tkzDrawPolygon(A,B,C)
```

9. Draw all points A, B, C, and D:

```
\tkzDrawPoints(A,B,C,D)
```

10. Print labels to the points and use options for positioning:

```
\tkzLabelPoints[below left](A)
\tkzLabelPoints(B,D)
\tkzLabelPoint[above](C){$C$}
```

11. Add the auxiliary circles of our intersection to the drawing with decent dots:

```
\tkzDrawCircle[dotted](A,B)
\tkzDrawCircle[dotted](B,A)
```

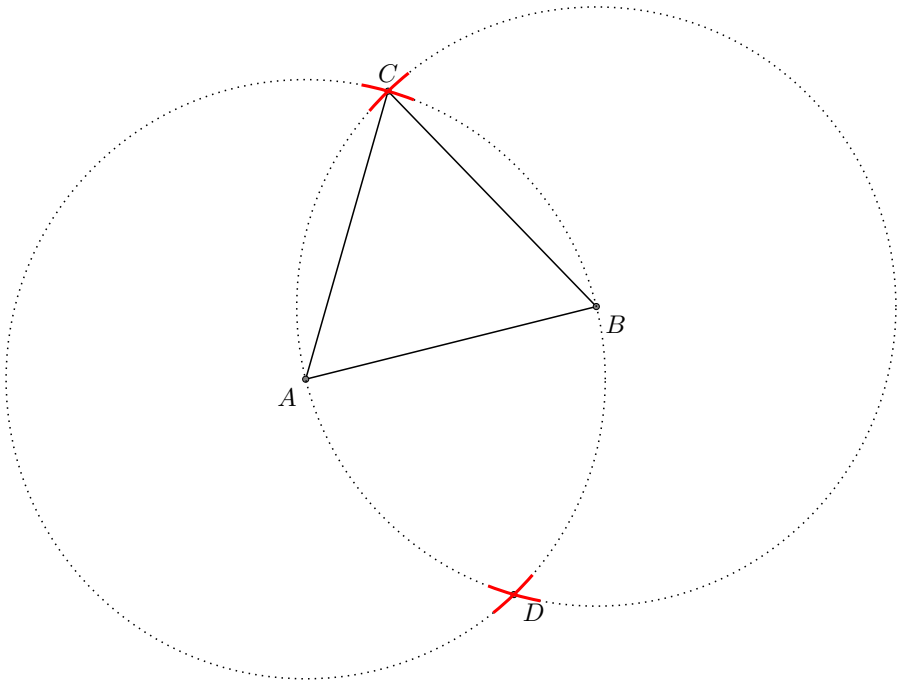
12. Add markers at C and D, like done with a compass, for illustration:

```
\tkzCompass[color=red, very thick](A,C)
\tkzCompass[color=red, very thick](B,C)
\tkzCompass[color=red, very thick](A,D)
\tkzCompass[color=red, very thick](B,D)
```

13. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

14. Compile, and have a look at our first result:



How it works....

Your first thought may be: *OMG! Another syntax!* However, it becomes clear which command belongs to the package when you simply prefix it with the `\tkz` string. This avoids possible name clashes just because another package can provide a command with the same name.

Besides that, optional arguments are given in square brackets. Often you can give general TikZ keys and values on such an occasion, like we did with red color and for very thick lines. Mandatory arguments are given in curly brackets, as usual. Coordinates, named points, line segments, and circles are provided using parentheses.

The basic documentation of the package is available by typing the `texdoc readme-tkz-euclide.txt` command in Command Prompt. When you type the `texdoc tkz-euclide` command, you will get documentation in French. Code as such is international, but without knowing French it will be a challenge. Besides that, the manual is an excellent reference, too much to repeat here. Hopefully, there will be an English version soon, accessible with `texdoc`. You may also visit `http://texdoc.net` and type `tkz-euclide` into the search box to see all available documentation versions.

A good general approach is:

- Define points with coordinates.
- Calculate further points, such as by using intersections or projections.
- Draw circles, lines in points in the order you like. For example, you could draw the points later than lines and circles to see them on top.
- Draw labels.
- Optionally, use TikZ commands to add anything.

Let's take a look at the basic commands. Unlike the reference manual, to easily digest the syntax, we will use sample values for explanation.

Defining points

You can define points with names using coordinates and even math formulas. These are some commands with their explanation:

- The `\tkzDefPoint(1,2){P}` defines a point with the name `P` at *x* coordinate `1` and *y* coordinate `2`. The `tkz-euclide` package uses cm as a length unit.
- The `\tkzDefPoints{1/2/A, 4/5/B}` defines the points `A` at (1,2) and `B` at (4,5). You can define even more points simultaneously.
- You can use calculations, such as the `\tkzDefPoint({2*ln(3) }, {sin(FPpi/2)){P}` with the syntax of the `fp` package (`texdoc fp`).
- You can optionally label it directly, such as by using the `\tkzDefPoint[label=left:$P$]( (1,2){P}`.

Calculating points

For later use, you can let the `tkz-euclide` package calculate points at intersections.

- The `\tkzInterLL(A,B)(C,D)` command intersects the line through the points `A` and `B` with the line through `C` and `D`.
- The `\tkzGetPoint{X}` command defines a point with the name `X` as a result of the last operation, like the preceding line intersection.
- The `\tkzInterLC(A,B)(C,D)` command intersects the line through `A` and `B` with the circle around `C` through `D`.
- The `\tkzInterCC(A,B)(C,D)` command intersects the circle around `A` and through `B` with the circle around `C` through `D`.
- The `\tkzGetPoints{X}{Y}` command defines points with the names `X` and `Y` as a result of the last operation, like for the preceding intersection.

There are further possible calculations:

- The `\tkzDefPointBy[translation = from A to B](X)` command defines a point as the translation of `X` by the line through `A` and `B`.
- The `\tkzDefPointBy[rotation = center M angle 90](X)` command rotates the point `X` around the center point `M` by `90` degrees.
- The `\tkzDefPointBy[rotation in rad = center M angle pi/2](X)` command rotates the point `X` around the center point `M` by `pi/2`, which is in Radian and means 90 degrees as in the preceding point.
- The `\tkzDefPointBy[symmetry = center M](X)` command reflects the point `X` at the point `M`.
- The `\tkzDefPointBy[reflection = over A--B](X)` command reflects the point `X` at the line through `A` and `B`.
- The `\tkzDefPointBy[projection = onto A--B](X)` command defines a point as the orthogonal projection from `X` to the line through `A` and `B`.
- The `\tkzDefPointBy[homothety = center M ratio 0.3](X)` command calculates a `homothety` count of the point `X` with center point `M` and ratio `0.3`.
- The `\tkzDefPointBy[inversion = center M through P](X)` command calculates an inversion of the point `X` with center point `M` and through the point `P`.

  Again, you can get the result of the calculation by the `\tkzGetPoint` command.

- The `\tkzDefPointsBy[operation](A,B,C)` command defines point using operations like we just saw. The results will be called A', B', and C'.

Drawing objects

You can draw objects such as points, lines, and circles, using one of these commands:

- The `\tkzDrawPoint(A)` command draws the point `A` as a filled circle.
- The `\tkzDrawPoints(A,B,C)` command draws a list of points, here `A`, `B`, and `C`.
- The `\tkzDrawLine(A,B)` command draws the line through the points `A` and `B` with an offset before `A` and after `B`. You can modify that offset by an option: the `\tkzDrawLine[add=0 and 1](A,B)` command starts exactly at `A` but adds 1 cm after `B`.
- The `\tkzDrawLines(A,B C,D E,F)` command draws lines through the points `A` and `B`, `C` and `D`, `E` and `F`, respectively. It understands the same offset as we just saw.
- The `\tkzDrawSegment(A,B)` command draws the line segment from point `A` to `B`.
- The `\tkzDrawSegments(A,B C,D)` command draws line segments from point `A` to point `B` and from `C` to `D`.
- The `\tkzDrawCircle(A,B)` command draws a circle around the point `A` through the point `B`.

All drawing commands understand TikZ options, such as in the following snippet:

```
\tkzDrawCircle[color=blue, fill=yellow, opacity=0.5](A,B)
```

Printing labels

You can print labels to objects using one of the following commands:

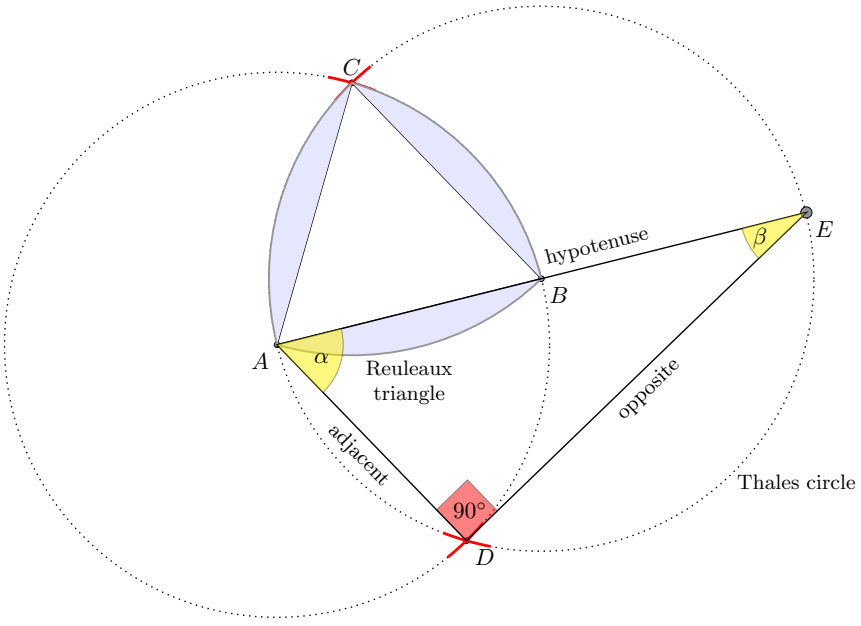
- The `\tkzLabelPoint(P){$P_1$}` command draws a label for the point `P` with subscript `1`.
- The `\tkzLabelPoints(A,B,C)` command draws labels for the points `A`, `B`, and `C`.
- The `\tkzLabelLine[left, pos=0.2](A,B){$L$}` command draws a label `L` left of the line through `A` and `B`, at the position `0.2`, so near the start.
- The `\tkzLabelSegment[above, pos=0.8](A,B){$S$}` command draws a label `S` the segment from `A` to `B`, at the position `0.8`, so near the end. `0.5` is the default position, which means the middle between `A` and `B`.
- The `\tkzLabelSegments[above](A,B C,D)` command labels the segments from `A` to `B` and from `C` to `D`.
- The `\tkzLabelCircle[draw, fill=yellow](A,B)(90){Circle of Apollonius}` command draws a rectangular node, filled with yellow color and the text "`Circle of Apollonius`", at a position of `90` degrees.

There's more...

Let us use some commands from the preceding example to extend our drawing. Add at the end, that means, right before the `\end{tikzpicture}` command:

```
\tkzInterLC(A,B)(B,A)
\tkzGetPoints{F}{E}
\tkzDrawPoints(E)
\tkzLabelPoints(E)
\tkzDrawPolygon(A,E,D)
\tkzMarkAngles[fill=yellow,opacity=0.5](D,A,E A,E,D)
\tkzMarkRightAngle[size=0.65,fill=red,opacity=0.5](A,D,E)
\tkzLabelAngle[pos=0.7](D,A,E){$\alpha$}
\tkzLabelAngle[pos=0.8](A,E,D){$\beta$}
\tkzLabelAngle[pos=0.5,xshift=-1.4mm](A,D,D){$90^\circ$}
\tkzLabelSegment[below=0.6cm,align=center,
  font=\small](A,B){Reuleaux\\triangle}
\tkzLabelSegment[above right,sloped,font=\small](A,E){hypotenuse}
\tkzLabelSegment[below,sloped,font=\small](D,E){opposite}
\tkzLabelSegment[below,sloped,font=\small](A,D){adjacent}
\tkzLabelSegment[below right=4cm,font=\small](A,E){Thales circle}
```

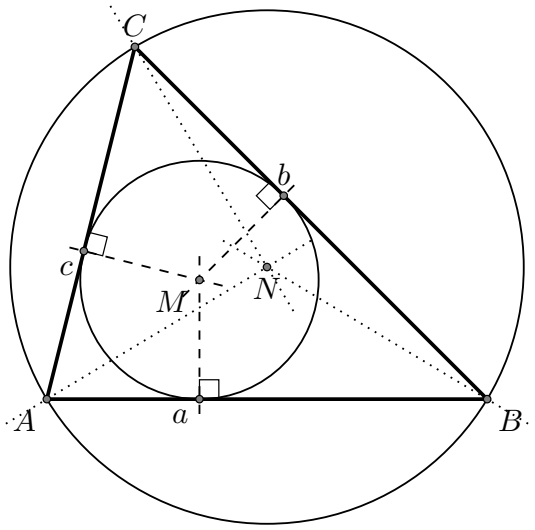
Compile the extended code, and take a look at the effect of those commands:



To see how easy it is to define circles tangential or by peripheral points, here's another small exercise using the new commands:

```
\documentclass[border=10pt]{standalone}
\usepackage{tkz-euclide}
\usetkzobj{all}
\begin{document}
\begin{tikzpicture}
  \tkzDefPoints{0/0/A, 5/0/B, 1/4/C}
  \tkzDefCircle[in](A,B,C)
  \tkzGetPoint{M}
  \tkzGetLength{r}
  \tkzDefCircle[circum](A,B,C)
  \tkzGetPoint{N}
  \tkzGetLength{R}
  \tkzDefPointBy[projection=onto A--B](M)
  \tkzGetPoint{a}
  \tkzDefPointBy[projection=onto B--C](M)
  \tkzGetPoint{b}
  \tkzDefPointBy[projection=onto A--C](M)
  \tkzGetPoint{c}
  \tkzDrawCircle[R](M,\r pt)
  \tkzDrawCircle[R](N,\R pt)
  \tkzDrawPolygon[very thick](A,B,C)
  \tkzDrawLines[dotted](N,A N,B N,C)
  \tkzDrawLines[dashed](M,a M,b M,c)
  \tkzMarkRightAngles(M,a,B M,b,C M,c,C)
  \tkzDrawPoints(A,B,C,M,N,a,b,c)
  \tkzLabelPoints[below left](A,M,a,c)
  \tkzLabelPoints[below right](B)
  \tkzLabelPoints[above](C,b)
  \tkzLabelPoints[below](N)
\end{tikzpicture}
\end{document}
```

This code produces the following drawing:



With this geometry package on top of TikZ, you can draw geometric constructions easily.

Doing calculations

Besides writing math, sometimes it's useful to actually calculate something.

We have several options:

- The `calc` package offers basic math with LaTeX, with lengths and counters
- The `fp` package provides fixed point arithmetic with high precision
- The `pgfmath` package belongs to the PGF/TikZ package and provides many functions and a good parser
- LuaLaTeX is a version of LaTeX which allows programming calculation in the programming language Lua

Here, let's work with the `pgfmath` package, as we already worked with TikZ, and it's better documented than the other options.

How to do it...

Follow these steps:

1. Start a document, load the `tikz` package, and begin your document without indentation at the beginning:

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
\noindent
```

2. In your text, use the `\pgfmathparse` command for calculating and the `\pgfmathresult` command for printing:

```
In a right-angled triangle the two shortest sides
got widths of 3 and 7, respectively. The longest side has a
width of \pgfmathparse{sqrt(3^2 + 7^2)}\pgfmathresult.
```

3. Use the `\pgfmathprintnumber` command for formatted printing:

```
The smallest angle is about \pgfmathparse{atan(3/7)}
\pgfmathprintnumber[precision=2]{\pgfmathresult} degrees.
```

4. End the document:

```
\end{document}
```

5. Compile, and look for the values:

In a right-angled triangle the two shortest sides got widths of 3 and 7, respectively. Then, the longest side has a width of 0.0. The smallest angle is about 23.2 degrees.

How it works....

There are basically two steps:

1. The `\pgfmathparse` command does parsing and calculating.
2. The `\pgfmathresult` command prints or uses the result.

The `\pgfmathprintnumber` command is another way of printing the result with a desired precision.

For the formula to be parsed, you can use the usual operation symbols, parentheses, and well-known functions, such as `sqrt` for square root; `ln` for natural logarithm; `min` and `max`; a lot of trigonometric functions such as `sin`, `cos`, and `tan`; and many more.

For details, please have a look at the `pgfmath` section in the PGF/TikZ manual. The next section shows how to get it.

If you would like to do high-precision calculations, I would switch to LuaLaTeX or to the `fp` package. The reason is that the precision of the `pgfmath` package is limited by TeX's internal capabilities. However, if you use it like it was intended, and this is for drawing, you may not need precisions in parts of micrometers.

See also

While the recipes in this chapter focused on a quick start, there are extensive reference manuals for mathematics with LaTeX:

- One is the `amsmath` manual, describing the package in detail. You can open it by typing the `texdoc amsmath` command in Command Prompt or online at `http://texdoc.net/pkg/amsmath`.
- An even more comprehensive document can be opened by typing the `texdoc mathmode` command or visiting `http://texdoc.net/pkg/mathmode`.
- We used TikZ a lot to visualize mathematics. The manual for it is available by typing the `texdoc tikz` command in Command Prompt and on `http://texdoc.net/pkg/tikz`.
- There's a section in the TikZ example gallery with a lot of sample math drawings with full code:

  `http://texample.net/tikz/examples/area/mathematics/`

- There's also a gallery for 2D and 3D plots, with an extensive math section here:

  `http://pgfplots.net/tikz/examples/area/mathematics/`

These websites are maintained by me. If you have questions regarding the examples, mathematics with LaTeX, or this book, you can put them to me in the LaTeX Community forum at `http://latex-community.org/forum`.

11
Science and Technology

This chapter covers the following topics:

- Typesetting an algorithm
- Printing a code listing
- Application in graph theory
- Writing physical quantities with units
- Writing chemical formulae
- Drawing molecules
- Representing atoms
- Drawing electrical circuits

Introduction

While the previous chapter was about mathematics, we now deal with other sciences, such as chemistry, physics, computer science, and technology such as electronics.

Most sciences heavily use mathematics, which we already covered in *Chapter 10*, *Advanced Mathematics*. So, this chapter will be kind of an overview, showing with specific recipes how LaTeX can be used in various fields.

Typesetting an algorithm

A very fundamental topic in computer science is the concept of an **algorithm**. That's a set of operations performed step by step. The purpose can be calculations or data processing, for example, sorting.

Algorithms can be visualized by a flow chart, which we did in *Chapter 9*, *Creating Graphics*. In this recipe, we will print an algorithm using pseudocode with syntax highlighting. Our example will show the calculation for a point in the Mandelbrot set.

How to do it...

We will use the `algorithmicx` package. For a better explanation, we split the work into many small steps. As with all examples, you can download the complete code from `http://latex-cookbook.net`, so you don't need to type it. At the end, you will see an image with the output. You can switch to the output image and back to the how-to, to see how we build the algorithm layout step by step. Here it goes:

1. As usual, start with a document class:

```
\documentclass{article}
```

2. Load packages you intend to use. In this case, we need the packages `dsfont` and `mathtools`:

```
\usepackage{dsfont}
\usepackage{mathtools}
```

3. Load these three algorithm packages:

```
\usepackage{algorithm}
\usepackage{algorithmicx}
\usepackage{algpseudocode}
```

4. Define the commands that you need and that are not yet provided by `algorithmicx`, in our case, a statement for local variables:

```
\algnewcommand{\Local}{\State\textbf{local variables: }}
```

5. Define any other macros you need. In our case, we will define a shortcut `\Let` command for repeated variable assignments. For better alignment on the left-hand side, we define a macro `\minbox`, which creates a box with a minimum width.

```
\newcommand{\minbox}[2]{%
  \mathmakebox[\ifdim#1<\width\width\else#1\fi]{#2}}
\newcommand{\Let}[2]{\State $ \minbox{1em}{#1} \gets #2 $}
```

6. Start the document:

```
\begin{document}
```

7. Open an `algorithm` environment:

```
\begin{algorithm}
```

8. Provide a caption and a label for cross-referencing:

```
\caption{Mandelbrot set}
\label{alg:mandelbrot}
```

9. Start an `algorithmic` environment with an option *n* for numbering every nth line. We choose `1` as this option, numbering every single line.

```
\begin{algorithmic}[1]
```

10. You can state requirements, if any:

```
\Require{$c_x, c_y, \Sigma_{\max} \in \mathds{R},
  \quad I \in \mathds{N}, \quad i_{\max} > 0,
  \quad \Sigma_{\max} > 0$}
```

11. Write down the function name with arguments:

```
\Function{mandelbrot}{$c_x, c_y, i_{\max},
          \Sigma_{\max}$}
```

12. Now we use our own macro to declare local variables:

```
\Local{$x, y, x^\prime, y^\prime, i, \Sigma$}
```

13. We initialize local variables, using the macro we just wrote:

```
\Let{x, y, i, \Sigma}{0}
```

14. We can add a comment to the line:

```
\Comment{initial zero value for all}
```

15. Now, we write down a `while` loop, which contains assignments like we just did:

```
\While{$\Sigma \leq \Sigma_{\max}$
       and $i < i_{\max}$}
  \Let{x^\prime}{x^2 - y^2 + c_x}
  \Let{y^\prime}{2xy + c_y}
  \Let{m}{x^\prime}
  \Let{y}{y^\prime}
  \Let{\Sigma}{x^2 + y^2}
\EndWhile
```

16. Add an if ... then conditional statement:

```
\If{$i < i_{\max}$}
  \State \Return{$i$}
\EndIf
```

17. Specify a return value, and end the function:

```
  \State\Return{0}
\EndFunction
```

18. End all open environments and the document:

```
  \end{algorithmic}
\end{algorithm}
\end{document}
```

19. Compile the document and take a look:

Algorithm 1 Mandelbrot set

Require: $c_x, c_y, \Sigma_{\max} \in \mathbb{R}, \quad i \in \mathbb{N}, \quad i_{\max} > 0, \quad \Sigma_{\max} > 0$

1: **function** MANDELBROT($c_x, c_y, i_{\max}, \Sigma_{\max}$)
2: **local variables:** x, y, x', y', i, Σ
3: $x, y, i, \Sigma \leftarrow 0$ ▷ initial zero value for all
4: **while** $\Sigma \leq \Sigma_{\max}$ and $i < i_{\max}$ **do**
5: $x' \leftarrow x^2 - y^2 + c_x$
6: $y' \leftarrow 2xy + c_y$
7: $m \leftarrow x'$
8: $y \leftarrow y'$
9: $\Sigma \leftarrow x^2 + y^2$
10: **end while**
11: **if** $i < i_{\max}$ **then**
12: **return** i
13: **end if**
14: **return** 0
15: **end function**

How it works...

The `algorithm` environment is a wrapper that allows the algorithm to float to a good position, just like figures and tables. So page breaks within algorithms are avoided and pages can be well filled. Furthermore, it supports captions and labels for cross-referencing.

The inner `algorithmic` environment does the specific typesetting. It supports commands that are commonly used in algorithm descriptions. We already saw some example, such as for `while` loops and for conditional statements. The full set of commands is described in the package manual, available by typing the `texdoc algorithmicx` command in Command Prompt or online at `http://texdoc.net/pkg/algorithmicx`.

There's more...

There's more than the pseudocode style. You can use the `algpascal` layout, which supports Pascal language syntax and even performs block indentation automatically. The `algc` layout is the equivalent for the C language; at the time of writing, it's still unfinished.

Experienced users may define their own command sets. This, together with existing layout features, is described in the package manual.

Printing a code listing

Documentation may contain code samples. The same applies to theses in computer science. While pseudocode of algorithms was covered in the previous recipe, we now like to typeset real code. To save space, we will use a small "hello world" program as an example.

How to do it...

We will use the `listings` package, which has been designed for that purpose. Follow these steps:

1. Start with any document class:
   ```
   \documentclass{article}
   ```
2. Load the `listings` package:
   ```
   \usepackage{listings}
   ```

3. Begin the document:

```
\begin{document}
```

4. Start an `lstlisting` environment with an option for the language:

```
\begin{lstlisting}[language = C++]
```

5. Continue with the code you would like to print:

```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main() {
  std::cout << ""Hello TeX world!"" << std::endl;
}
```

6. End the `lstlisting` environment and the document:

```
\end{lstlisting}
\end{document}
```

7. Compile the document and have a look:

```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main()
{
    std::cout << "Hello TeX world!" << std::endl;
}
```

How it works...

The basic usage is simple:

1. Load the package.
2. Surround each code listing by a `lstlisting` environment, optionally specifying a language as we did.

Refer to the manual for the list of supported languages, which is still growing today. You may define your own language style too or find one for your favorite language on the Internet. You could simply ask in a forum for it. In the next and final chapter, we will visit a LaTeX Internet forum.

Commands and environments of the listings package start with the `\lst` prefix, so there's no name collision with another package.

By just calling one command, you can customize the appearance of all your listings:

```
\lstset{key1 = value1, key2 = value2}
```

It's an extensive `key=value` interface with a lot of keys. Let's take a look at how to use it, taking especially useful keys.

Modify the preceding example in this way:

1. Add the `xcolor` package to your document preamble:

   ```
   \usepackage{xcolor}
   ```

2. Load the `inconsolata` package for an excellent typewriter font:

   ```
   \usepackage{inconsolata}
   ```

3. Define frequently used macros, such as for the programming language logo, for consistent appearance:

   ```
   \newcommand{\Cpp}{C\texttt{++}}
   ```

4. After `\usepackage{listings}`, insert settings via `key=value`:

   ```
   \lstset{
     language         = C++,
     basicstyle       = \ttfamily,
     keywordstyle     = \color{blue}\textbf,
     commentstyle     = \color{gray},
     stringstyle      = \color{green!70!black},
     stringstyle      = \color{red},
     columns          = fullflexible,
     numbers          = left,
     numberstyle      = \scriptsize\sffamily\color{gray},
     caption          = A hello world program in \Cpp,
     xleftmargin      = 0.16\textwidth,
     xrightmargin     = 0.16\textwidth,
     showstringspaces = false,
     float,
   }
   ```

5. Thanks to the settings, you can now use `\begin{lstlisting}` without arguments. Compile your modified example, and look at the change:

Listing 1: A hello world program in C++

```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main()
{
    std::cout << "Hello TeX world!" << std::endl;
}
```

There's more...

Like the standard LaTeX `verbatim` environment and the `\verb` command, `lstlisting` also has a companion for including small pieces of code inline: the command `\lstinline`. Write it like this:

```
Use \lstinline!#include <iostream>! for including i/o streams.
```

Instead of the exclamation mark, you could use any character as delimiter, as long as it does not appear in the code snippet.

You can store longer listings in external files. Instead of the standard `\input` command, use this:

```
\lstinputlisting[options]{filename}
```

You can use the same options as for the `lstlisting` environment. For example, note this:

```
\lstinputlisting[firstline=4, lastline=10]{filename}
```

It includes only lines `4` to `10`. This way, you can divide longer listings to explain them with normal text in between.

You can generate a list of listings with their captions using the command `\lstlistoflistings`.

Application in graph theory

In graph theory, models and drawings often consist mostly of vertices, edges, and labels. So, it may be possible to use a simpler language to generate a diagram of a graph.

How to do it...

The `tkz-graph` package offers a convenient interface. We will create a diagram with it as follows:

1. Begin with any document class:
   ```
   \documentclass{standalone}
   ```
2. Load the `tkz-graph` package:
   ```
   \usepackage{tkz-graph}
   ```
3. Specify a basic style:
   ```
   \GraphInit[vstyle = Shade]
   ```
4. Customize element styles as desired using standard TikZ syntax:
   ```
   \tikzset{
     LabelStyle/.style = { rectangle, rounded corners,
                           draw, minimum width = 2em,
                           fill = yellow!50,
                           text = red, font = \bfseries },
     VertexStyle/.append style = { inner sep=5pt,
                                   font = \Large\bfseries},
     EdgeStyle/.append style = {->, bend left} }
   ```
5. Begin the document:
   ```
   \begin{document}
   ```
6. Start a `tikzpicture` environment:
   ```
   \begin{tikzpicture}
   ```
7. Set the distance between vertices. The default is `1` for 1 cm.
   ```
   \SetGraphUnit{5}:
   ```

8. Declare a first vertex B:

```
\Vertex{B}
```

9. Set a vertex A to the west (WE) and C to the east (EA):

```
\WE(B){A}
\EA(B){C}
```

10. Draw edges between the vertices:

```
\Edge[label = 1](A)(B)
\Edge[label = 2](B)(C)
\Edge[label = 3](C)(B)
\Edge[label = 4](B)(A)
```

11. Add loops, which are edges from a vertex to itself:

```
\Loop[dist = 4cm, dir = NO, label = 5](A.west)
\Loop[dist = 4cm, dir = SO, label = 6](C.east)
```

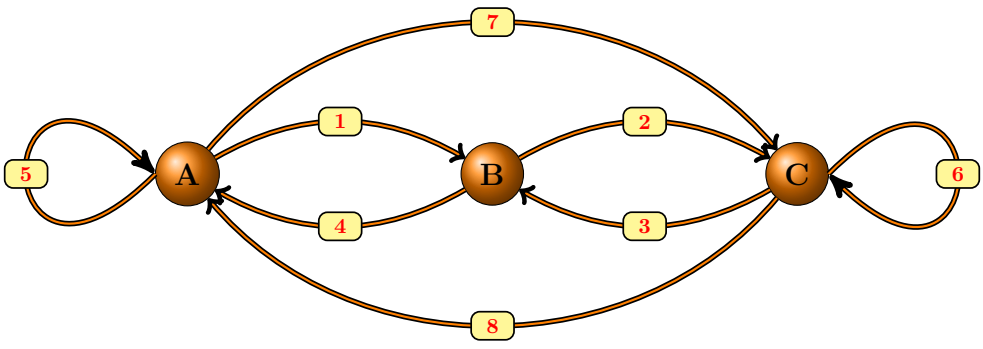
12. Adjust the bend angle of the edges for the final two wider edges:

```
\tikzset{EdgeStyle/.append style = {bend left = 50}}
\Edge[label = 7](A)(C)
\Edge[label = 8](C)(A)
```

13. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

14. Compile and have a look:



How it works...

As with other recipes in this book, the basic procedure is:

1. Define styles
2. Position vertices
3. Add edges
4. Repeat if needed

For step 1, there are basic styles set by `vstyle` as we did earlier:

- `Empty`: This gives simple vertices without circle or ball and simple lines as edges.
- `Classic`: This gives vertices that are filled circles around the vertex label and edges that are simple lines.
- `Normal`: This gives vertices that are circles and edges that are simple lines.
- `Art`: This gives vertices that are shaded balls and edges colored in orange.
- `Shade`: This gives vertices that are shaded balls and edges that are thick lines with borders.
- Further variations are `Simple`, `Hasse`, `Dijkstra`, `Welsh`, and `Shade Art`.

They are pictured in the manual, though you could simply try the names with your diagram. You can open the manual by typing `texdoc tkz-graph` in Command Prompt, or online at `http://texdoc.net/pkg/tkz-graph`.

For step 2, there's a simple syntax for adding vertices:

```
<direction>(B){A}
```

Here, `<direction>` can be:

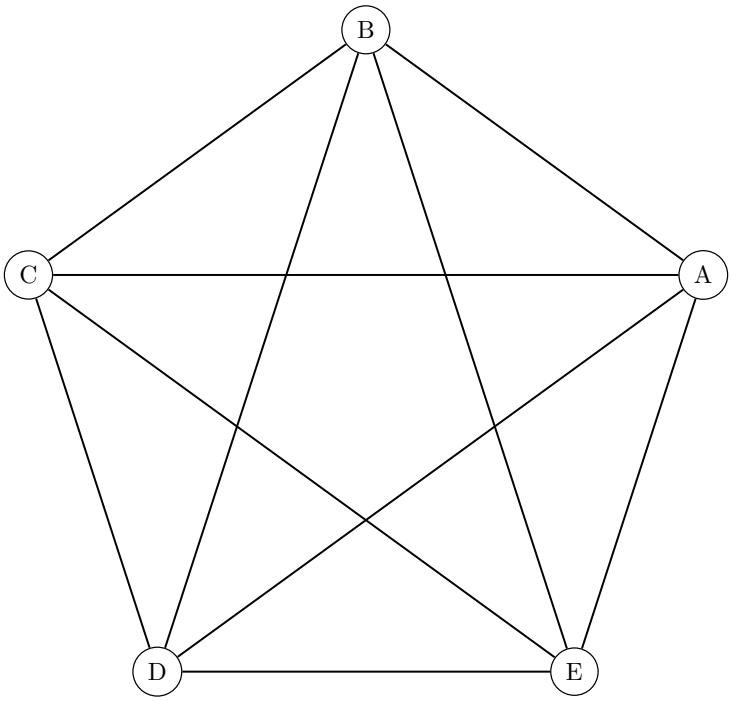
- `\EA` for setting b to the east of A
- `\WE` for doing that to the west
- `\NO` for doing that to the north
- `\SO` for doing that to the south
- `\NOEA`, `\NOWE`, `\SOEA`, and `\SOWE` as combinations

For step 3, edges are done by using `\Edge[options](vertex)(vertex)`, as we just saw. This together already allows quickly setting up graphs.

There are plural command versions, making it even more compact, for example, for a set of vertices and a chain of edges. This is illustrated in this small example:

```
\documentclass{standalone}
\usepackage{tkz-graph}
\begin{document}
\begin{tikzpicture}[rotate=18]
  \SetGraphUnit{5}
  \GraphInit[vstyle=Normal]
  \Vertices{circle}{A,B,C,D,E}
  \Edges(A,B,C,D,E,A,D,B,E,C,A)
\end{tikzpicture}
\end{document}
```

It also demonstrates the `Normal` style:



Writing physical quantities with units

In contrast to pure mathematics, in the natural sciences such as chemistry and physics, and in engineering, we use units in addition to quantities. We need to distinguish units from variables.

Let's take a sample. We shall write a formula that multiplies the speed *s* of one meter per second by the factor *m*. A straightforward way of doing this could be like so:

```
\( m \cdot s = m \cdot 1 m s^{-1} \)
```

The LaTeX standard output would be:

$$m \cdot s = m \cdot 1ms^{-1}$$

What on earth? It's pretty tempting to mathematically simplify the right side, which could be interpreted as square meters per second or as millimeters per second. Or multiply both sides by s, which may mean speed (s) or seconds. Units and variables look very similar. Furthermore, our space between *1* and *m* has been lost. For smart writing that fits common standards, we may require the following:

- Units to be written upright to distinguish them from italic math variables
- There to be a small space between a quantity and a unit
- The appearance to be customizable without changing formulas, such as in a case when a journal would require a different style
- Semantic writing such as meters instead of *m* and seconds instead of s, which would be a helpful addition
- Smart parsing of numbers in quantities, which may be beneficial
- Striking out or highlighting would be great for explaining a calculation

Can all of this be achieved? Yes, it can!

How to do it...

The `siunitx` package provides ways to comply with international conventions regarding unit systems and is customizable to match typographic styles.

Let's correct the formula we just saw and apply modifications using these steps:

1. Start with any document class:
   ```
   \documentclass{article}
   ```
2. Load the `siunitx` package:
   ```
   \usepackage{siunitx}
   ```
3. Start the document:
   ```
   \begin{document}
   ```
4. Write the formula from earlier, but this time use the command `\SI{quantity}{units}` like this:
   ```
   \( m \cdot s = m \cdot \SI{1}{\m\per\s} \)
   ```
5. End the document for now:
   ```
   \end{document}
   ```
6. Compile, and take a look:

$$m \cdot s = m \cdot 1\,\mathrm{m\,s^{-1}}$$

7. You can also use longer natural names for units. This is achieved by:
   ```
   \( m \cdot s = m \cdot \SI{1}{\meter\per\second} \)
   ```
8. Let's modify the reciprocal units. Add to your preamble after loading `siunitx`:
   ```
   \sisetup{per-mode = symbol}
   ```
9. Compile to see the difference:

$$m \cdot s = m \cdot 1\,\mathrm{m/s}$$

10. If you would like to emphasize changes, you can do this using the `cancel` and `color` packages. Add them to your preamble:

```
\usepackage{cancel}
\usepackage{color}
```

11. Let's test this together with scientific exponential notation. So, change your formula line as follows:

```
\( m \cdot s = m \cdot
  \SI{1e-3}{\cancel\m\highlight{red}\km\per\s} \)
```

12. Compile to see the latest result:

$$m \cdot s = m \cdot 1 \times 10^{-3} \cancel{\mathrm{m}}\,\mathrm{km}/\mathrm{s}$$

How it works...

The command `\SI{quantity}{units}` does two jobs:

- It parses the quantity in its first argument and formats the numbers. It understands complex numbers and exponential notations. In the final formatting, superfluous spaces are removed. Large numbers consisting of many digits are grouped into blocks of three by a thin space.
- It parses the units and typesets them in a proper way, such as with a thin space between the quantity and the unit.

So, actually, `\SI` combines two commands, which you also can use directly:

- `\num{numbers}` parses numbers in the argument and formats them properly.
- `\si{units}` typesets the units. For example, `\si{\kilo\gram\meter\per\square\second}` or shorter `\si{\kg\m\per\square\s}` gives:

$$\mathrm{kg\,m/s^2}$$

The package implements the basic set of SI standardized units by macros, including derived units. So you can type `\meter` and `\metre`, `\gram`, and so on, but also derived units such as `\newton`, `\watt`, `\hertz`, and many more. Even non-SI units are supported, such as `\hour` or `\hectare`. The package also supports common prefixes such as `\kilo`, `\mega`, and `\micro`. The full list of features is written down in the excellent manual. You can access it by typing the `texdoc siunitx` command in Command Prompt or by visiting `http://texdoc.net/pkg/siunitx`.

Writing chemical formulae

Chemical formulae and equations have a different style compared to mathematical formulae and equations. For example:

- Letters mean atomic symbols and are written upright, unlike italic math variables
- Numbers are commonly used in subscripts, indicating the number of atoms
- We use a lot of subscripts and superscripts, and they should be aligned properly
- We need also left subscripts and superscripts
- We need special symbols such as for bonds and arrows for chemical equations

With basic LaTeX, it's hard to achieve all of this. Let's find a better way.

How to do it...

We will use the `chemformula` package. We will boldly go ahead and type some chemical stuff to see how it works. The LaTeX output will follow, so you may look ahead line by line if you like. Let's start:

1. Begin with a document class of your choice, such as `scrartcl` of KOMA-Script:
   ```
   \documentclass{scrartcl}
   ```
2. Load the `chemformula` package:
   ```
   \usepackage{chemformula}
   ```
3. Start the document:
   ```
   \begin{document}
   ```
4. We begin with an unnumbered section to verify that formulae work in headings. We use the command `\ch` and give atoms and numbers as argument, directly, without _ and ^.
   ```
   \section*{About \ch{Na2SO4}}
   \ch{Na2SO4} is sodium sulfate.
   ```

5. Also electric charges of ions are simply written without _ and ^:

```
It contains \ch{Na+} and \ch{SO4^2-}.
```

6. Adducts can be written with a star or a dot. Numbers can be detected as being a stoichiometric factor. But leave a blank as separator:

```
\ch{Na2SO4 * 10 H2O} is a decahydrate.
```

7. We can use chemical formulae also in math mode. So we write a displayed and centered equation with a forward arrow, also called reaction arrow, with `->`:

```
\[
  \ch{Na2SO4 + 2 C -> Na2S + 2 CO2}
\]
```

8. We can have it numbered too, like math equations. This time we use an equilibrium arrow, generated by `<=>`:

```
\begin{equation}
  \ch{Na2SO4 + H2SO4 <=> 2 NaHSO4}
\end{equation}
```

9. If a number is on the left of an atom, it acts as a left subscript. But we can clearly indicate the meaning using _ and ^ before an atom, such as for isotopes:

```
\section*{Isotopes}
\ch{ ^{232}_{92}U140} is uranium-232.
```

10. Common bonds with one, two, or three lines are written with `-`, `=`, or `+`, respectively. We can see this in a list of hydrocarbons:

```
\begin{itemize}
  \item \ch{H3C-CH3} is ethane,
  \item \ch{H2C=CH2} is ethylene,
  \item \ch{H2C+CH2} is ethyne.
\end{itemize}
```

11. That's enough for now. Let's end the document:

```
\end{document}
```

12. Compile to see what we have done:

About Na_2SO_4

Na_2SO_4 is sodium sulfate. It contains Na^+ and SO_4^{2-}. $Na_2SO_4 \cdot 10\,H_2O$ is a decahydrate.

$$Na_2SO_4 + 2\,C \longrightarrow Na_2S + 2\,CO_2$$

$$Na_2SO_4 + H_2SO_4 \rightleftharpoons 2\,NaHSO_4 \tag{1}$$

Isotopes

$^{232}_{92}U_{140}$ is uranium-232.

Hydrocarbons

- $H_3C{-}CH_3$ is ethane,
- $H_2C{=}CH_2$ is ethylene,
- $H_2C{\equiv}CH_2$ is ethyne.

How it works...

The input syntax is as easy as possible in the most natural way:

- Atoms are typed as letters.
- Numbers are written as subscript by default, indicating the number of atoms in the formula.
- Stoichiometric numbers, which mean numbers of molecules, are written before the molecule with a space between.

This not only makes typing easier but also makes copy and paste from PDF or Word documents or from the Internet very easy. The most common bonds are written by:

- \- for a single bond
- = for a double bond
- \+ for a triple bond

Reaction arrows are as follows:

- `->` and `<-` draw simple arrows to the right or to the left
- `-/>` and `</-` draw broken arrows to the right or to the left (does not react)
- `<->` draws a resonance arrow (arrows with tip at the left and at the right)
- `<>` gives a right arrow at the top and a left arrow under it
- `<=>` draws an equilibrium arrow (half of an arrow tip at each side)
- `<=>>` draws an equilibrium arrow with tendency to the right, so the top arrow to the right is larger
- `<<=>` draws an equilibrium arrow with tendency to the left, so the lower arrow to the left is larger

You can set math, chemical expressions, or text above or below arrows, like so:

```
<=>[\text{above}] [\text{below}]
```

There are more arrow types and features which you can read about in the package manual. As usual, you can open it by typing the `texdoc chemformula` command in Command Prompt or download it from `http://texdoc.net/pkg/chemformula`.

There's more...

The `mhchem` package can be used in a very similar way. There are some differences, explained in the `chemformula` manual. The newer `chemformula` package was intended to bring some improvements. Furthermore, it has been developed as part of the `chemmacros` bundle, which brings even more features for chemical notation.

There's a nearly complete list of packages for chemistry with TeX, together with descriptions, at `http://www.mychemistry.eu/known-packages/`.

There's another package list available on CTAN: `http://ctan.org/topic/chemistry`.

We pick another great package for our next recipe.

Drawing molecules

In the previous recipe, we already encountered formulae for molecules. Now let's see how to draw them. This means drawing a group of atoms and connecting them by lines of various kinds.

How to do it...

This rather complex seeming task becomes manageable using the `chemfig` package. It provides a compact syntax for drawing molecules. Let's draw some:

1. Start with any document class:

```
\documentclass{article}
```

2. Load the `chemfig` package:

```
\usepackage{chemfig}
```

3. Let's write molecules in a table. For this, we stretch the rows a bit and start a `tabular` environment with a right-aligned column and a left-aligned one.

```
\renewcommand{\arraystretch}{1.5}
\begin{tabular}{rl}
```

4. For molecules, use the `\chemfig` command. Write atoms as letters and a single bond using a dash:

```
  Hydrogen: & \chemfig{H-H} \\
```

5. Write a double bond using an equal-to sign:

```
  Oxygen:   & \chemfig{O=O} \\
```

6. For a triple bond, use a tilde:

```
  Ethyne:   & \chemfig{H-C~C-H}
```

7. End the table, and leave some space:

```
\end{tabular}
\qquad
```

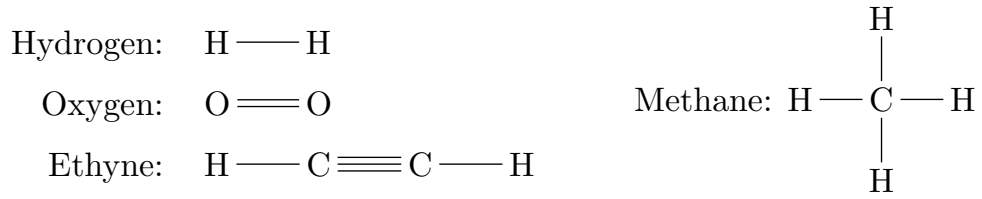
8. Enclose branches in parentheses. Options can be inserted using square brackets separated by commas. The first option is an angle. Here you can state multiples of 45 degrees but also arbitrary angles, which we shall see later. The second option is a factor for interatomic distance. We will define `0.8` for a more compact drawing. Use this to draw methane:

   ```
   Methane: \chemfig{[,0.8]C(-[2]H)(-[4]H)(-[6]H)-H}
   ```

9. Finish the document:

   ```
   \end{document}
   ```

10. Compile and take a look at the drawings:



How it works...

The `chemfig` package internally uses TikZ for drawing. It automatically takes care of the bounding box, so the drawing does not overlap other text. Advanced users could even insert TikZ code.

The main command is `\chemfig`; it takes an argument consisting of the following:

- Letters for atoms
- Symbols for bonds, such as `-`, and `~` for simple, double, and triple bonds, respectively
- Options in square brackets, separated by commas
- Branches of atoms and bonds within parentheses

Let's have a look at the options. These are the fields:

```
[angle,distance factor,number of departure atom,number of arrival
atom,TikZ options]
```

The angle can be one of the following:

- An integer representing a multiple of 45 degrees, such as `[2]` for 90 degrees
- An absolute angle in degrees, indicated by a double colon, such as `[:60]` for 60 degrees
- A relative angle in degrees, indicated by two colons, such as `[::30]` for 30 degrees in relation to the previous bond
- Positive and negative numbers are allowed

The angle is automatically reduced to the range between 0 and 360 degrees.

A branch in parentheses means that you can open a path by an opening parenthesis, define it like we just saw, and finish it using a closing parenthesis. Then, you are at the same position as when you started the branch.



In complex molecules, find the longest chain and draw it. Then add the branches. Using relative angles allows easy rotating of the whole molecule.

There's more...

There are further features we should take a look at.

Drawing rings

Rings in molecules are often drawn as regular polygons. They can be drawn using this syntax:

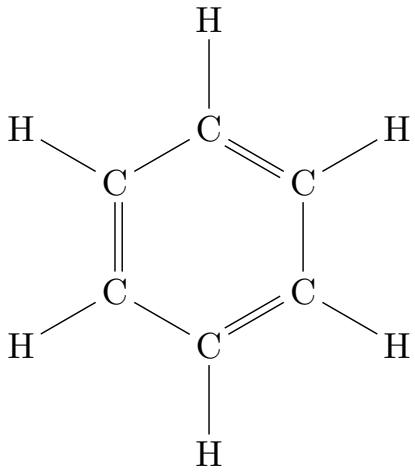
```
atom*n*(code)
```

Here, `n` means the number of sides of the polygon, and in parentheses is the `chemfig` code which goes into the ring, written like a linear structure.

The famous benzene ring with all atoms can be drawn like this:

```
\chemfig{C*6((-H)-C(-H)=C(-H)-C(-H)=C(-H)-C(-H)=)}
```

This line gives us the following output:



Naming molecules

You can write the name of a molecule underneath it. This is done as follows:

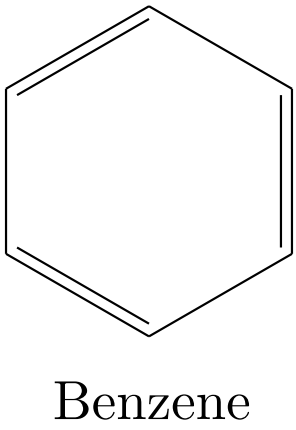
```
\chemname[distance]{\chemfig code}{name}
```

An optional value defines the distance to the baseline of the molecule. The default value is 1.5 ex.

For example, this is how we put the name under the carbon skeleton of the benzene ring:

```
\chemname{\chemfig{*6(=-=-=-)}}{Benzene}
```

We get the following result:



Benzene

Using building blocks

With LaTeX as the macro language, it's no surprise that `chemfig` also offers a similar functionality. As with the `\newcommand` macro, you can define submolecules for repeated use, like so:

```
\definesubmol{name}{code}
```

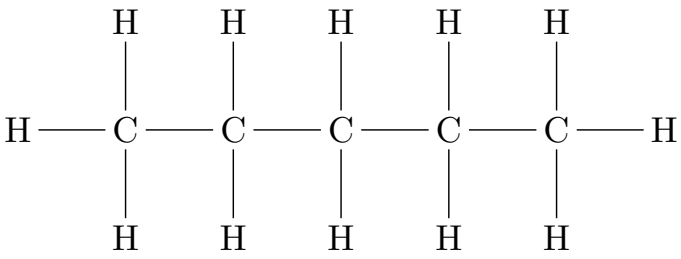
For example, here's how to define a molecule part with a carbon atom and two hydrogen atoms:

```
\definesubmol{C}{-C(-[2]H)(-[6]H)}
```

We can use this to draw Pentane:

```
\chemfig{H!C!C!C!C!C-H}
```

This short code generates a pretty big molecule:



Applying TikZ options

We can pass TikZ options to molecule drawings. `\chemfig` understands the two optional arguments are in this same line, each one is enclosed in square brackets:

```
\chemfig[options for tikzpicture][options for nodes]{code}
```

The options in the first argument are applied to the whole `tikzpicture` environment of the molecule. The options in the second argument would be added to the style of every node.

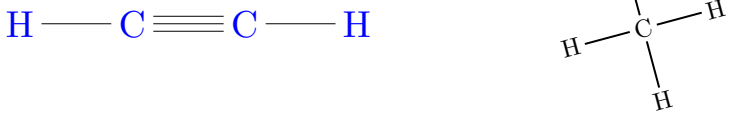
So, these options scale the whole picture and make only the nodes blue:

```
\chemfig[scale=1.5,transform shape][color=blue]{H-C~C-H}
```

These options produce thick lines and rotate the nodes by 15 degrees:

```
\chemfig[thick][rotate=15]{C(-[2]H)(-[4]H)(-[6]H)-H}
```

This would be the output of both lines:



Using ready-drawn carbohydrates

Though `chemfig` makes drawing easier, writing down complex molecules can still be laborious if you get a lot of them, such as in lecture notes or exam sheets about carbohydrates. Luckily, we don't need to reinvent the wheel all the time.

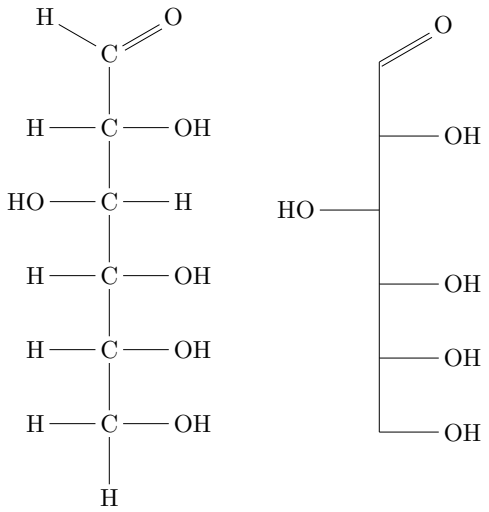
The `carbohydrate` package provides a lot of chemfig-drawn carbohydrates for you to use. It contains trioses, tetroses, pentoses, and hexoses, and various models: the Fischer model (full and skeleton), the Haworth model, and the chain model. You can draw them as ring isomer and chain isomer.

You can download the package at `https://github.com/cgnieder/carbohydrates`. There you can also find the manual.

Let's have a look at how easy it becomes, for example with glucose:

```
\glucose[model=fischer,chain]\quad
\glucose[model={fischer=skeleton},chain]
```

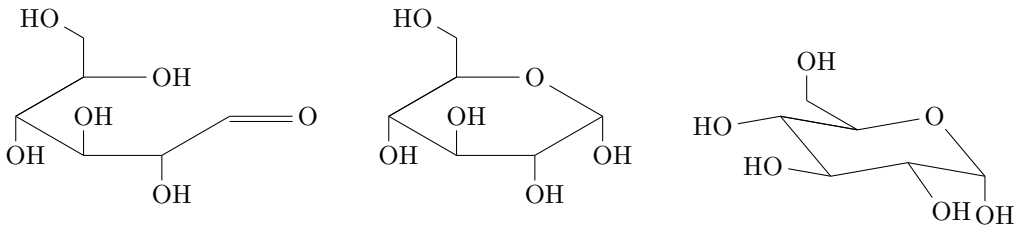
This draws the Fischer models; the skeleton version is without showing H and C atoms:



Now let's try glucose with other models:

```
\glucose[model=haworth,chain]\hfill
\glucose[model=haworth,ring]\hfill
\glucose[model=chair,ring]
```

We get the following structures:



Already implemented are these structures:

- `\glycerinaldehyde` (triose)
- `\erythrose`, `\threose` (tetroses)
- `\ribose`, `\arabinose`, `\xylose`, `\lyxose` (pentoses)
- `\allose`, `\altrose`, `\glucose`, `\mannose`, `\gulose`, `\idose`, `\galactose`, `\talose` (hexoses)

The package manual tells you all the details about options and usage. Its author also plans to develop the package further.

Representing atoms

Now that we can draw molecules, how about digging deeper? Can we draw atoms? Sure!

How to do it...

We will use a package named after the famous physicist Niels Bohr. That's `bohr`.

1. Start with a document class.

   ```
   \documentclass{article}
   ```

2. Load the `bohr` package:

   ```
   \usepackage{bohr}
   ```

3. Start the document:

```
\begin{document}
```

4. Use the command `\bohr{number of electrons}{element name}`, for Fluorine:

```
\bohr{10}{F}
```

5. We can modify the nucleus radius this way:

```
\setbohr{nucleus-radius=1.5em}
```

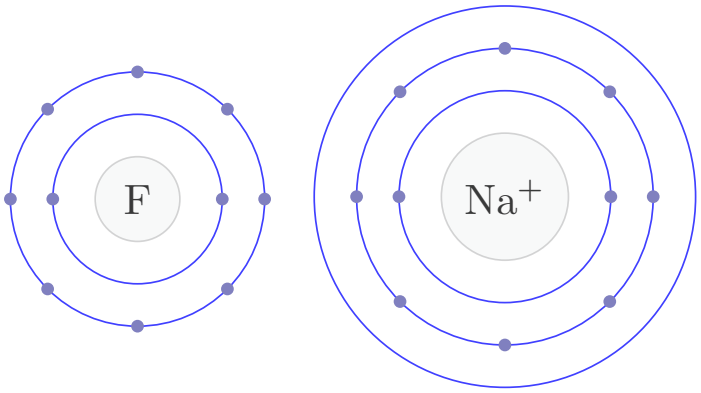
6. Now we have more space in the center for an ion symbol. This time we use the command with an optional argument for the number of electron shells since we need an empty one. We need to do that in square brackets. We will display a Sodium ion:

```
\bohr[3]{10}{$\mathrm{Na^+}$}
```

7. That's all for now! End the document:

```
\end{document}
```

8. Compile to see the result:



How it works...

It was quite simple. Nevertheless, I wanted to show you how easy it can be to write about science today.

After loading the package, we just needed this command:

```
\bohr[number of shells]{number of electrons}{element name}
```

The command `\setbohr` provides a `key=value` interface for customization. To not stretch the patience of non-physicists and non-chemists, we will omit the long list of optional parameters. You can read all customization details in the manual, which you can open by typing the `texdoc bohr` command in Command Prompt or find online at `http://texdoc.net/pkg/bohr`.

Drawing electrical circuits

Electrical documents, especially for learners, can contain a lot of formulas. So LaTeX, with its strengths in math typesetting, is a very good choice for writers. Electrical units are easily written complying with standards using the `siunitx` package, as we saw in a previous recipe in this chapter.

It seems natural to also draw circuits directly within LaTeX. In contrast to including external drawings, native LaTeX drawings can have annotations that perfectly match the text with font and styles.

So, this recipe will deal with drawing circuits. Our goal is to create a circuit with common electrical components such as a resistor, a diode, a capacitor, a bulb, and more.



This is just a sample illustration, so `don't` try building this one with real components at home.

How to do it...

The TikZ package provides several libraries for drawing electrical and logical circuits. We will choose one following the IEC norm. As usual, the code can be downloaded from the publisher's website or from `http://latex-cookbook.net`. Let's take a look at the following steps:

1. Start with a document class. For our sample, let's choose the `standalone` class which generates a PDF file matching the size of our drawing:

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `tikz` package:

   ```
   \usepackage{tikz}
   ```

3. Load the `circuits.ee.IEC` library, which provides IEC norm symbols:

   ```
   \usetikzlibrary{circuits.ee.IEC}
   ```

4. Start the document:

   ```
   \begin{document}
   ```

5. Begin a TikZ picture. Since we will provide options, add an opening square bracket:

```
\begin{tikzpicture}[
```

6. State the desired style as an option:

```
  circuit ee IEC,
```

7. Choose a width for `x` and `y` units in the drawing:

```
  x = 3cm, y = 2cm,
```

8. Adjust the style for annotations. Here we choose a smaller font size:

```
  every info/.style = {font = \scriptsize},
```

9. You can change the appearance of a symbol. The default diode is without a fill. With this option we can choose a diode with a black fill:

```
  set diode graphic = var diode IEC graphic,
```

10. Similarly for a switch contact, we will choose one with a small circle:

```
  set make contact graphic = var make contact IEC graphic,
```

11. We need to close the options for the TikZ picture by a square bracket:

```
  ]
```

12. At first, we will draw six contact points, three in a row, in two rows. It's easy to use a `for` loop for this:

```
  \foreach \i in {1,...,3} {
    \node [contact] (lower contact \i) at (\i,0) {};
    \node [contact] (upper contact \i) at (\i,1) {};
  }
```

13. As we defined the contacts having names, given in parentheses in the preceding code snippet, we can refer to them using `upper contact 1`, `lower contact 3`, and so on. So, we will connect the upper-left contact and the lower-left contact with a line with a diode in the middle:

```
  \draw (upper contact 1) to [diode] (lower contact 1);
```

14. We stated the component name as an option to the path. We can do the same for a capacitor:

```
  \draw (lower contact 2) to [capacitor] (upper contact 2);
```

15. The component keys can have options. So we draw a line with a resistor, which has an electrical resistance of 6 ohm, with that value as an annotation:

```
  \draw (upper contact 1) to [resistor = {ohm = 6}]
        (upper contact 2);
```

16. Annotations can be different. Here we use a symbol for an adjustable resistor:

```
\draw (lower contact 2) to [resistor = {adjustable}]
      (lower contact 3);
```

17. We can have even more options. Useful options are `near start` and `near end` for positioning two components on a line:

```
\draw (lower contact 1) to [
         voltage source = {near start,
                           direction info = {volt = 12}},
         inductor = {near end}]
      (lower contact 2);
```

18. Let's do the same for an open contact and a battery, with some text as annotation:

```
\draw (upper contact 2) to [make contact = {near start},
                            battery = {near end,
                                       info = {loaded}}]
      (upper contact 3);
```

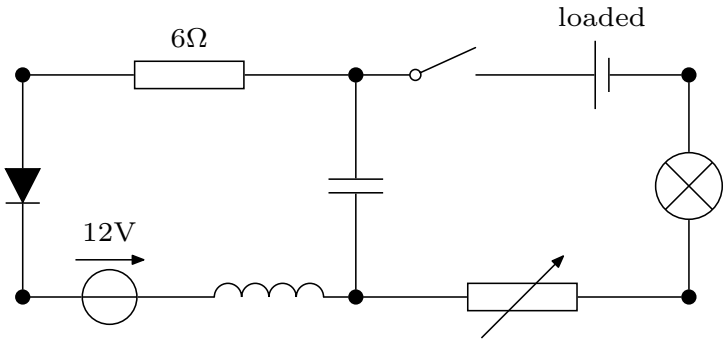
19. Let's finish with a bulb. We will make it a bit bigger than the default:

```
\draw (lower contact 3) to [bulb =
        {minimum height = 0.6cm}]
      (upper contact 3);
```

20. End the picture and the document:

```
\end{tikzpicture}
\end{document}
```

21. Compile and have a look at the circuit:



How it works...

The TikZ manual contains a reference of the circuit libraries with symbols and options. We cannot list all of them here. Basically, we did the following:

1. Loading the required library and define styles, either as options to the picture or globally via the `\tikzset` command.
2. Positioning contacts and other nodes. This can be done by pure coordinates, eased by a loop, or using the `positioning` library. The `matrix of nodes` parameter would be another option.
3. Drawing lines between the nodes. Use to-paths, which take components as options.

Components can have further options, such as for additional information (`info above`), for positioning (`near start` and `near end`), or color and size.

In our example, we used long names and a lot of spacing to keep the code readable. This is recommended especially when drawings become more complex.

You can open the TikZ manual by typing the `texdoc pgfmanual` or `texdoc tikz` commands in Command Prompt. You may also read it online by visiting `http://texdoc.net/pkg/tikz`.

See also

While writing about science and technology is often similar to writing maths, there are packages supporting field-specific writing conventions. We saw some for chemistry, and the support package for physics is the package with the same name. As usual, you can find documentation at `http://texdoc.net/pkg/physics` and via the command `texdoc physics`.

Today it is exciting that we can use LaTeX for drawing scientific figures. A lot of examples are available. I maintain a TikZ gallery of drawings with full code, which is also sorted by the fields under the sciences.

So, you can look through graphics made with LaTeX by browsing the gallery:

- `http://texample.net/tikz/examples/area/physics/` shows about 50 examples, such as 3D atom clusters, energy level diagrams, drawings for optics, mechanics, astronomy, and more
- `http://texample.net/tikz/examples/area/chemistry/` shows 15 illustrations for chemistry, such as a periodic table of elements, and is growing and is growing
- `http://texample.net/tikz/examples/area/computer-science/` contains about 40 drawings of networks, database topics, protocols, algorithms, and more

And there are more areas to browse, such as biology, astronomy, economics, and more.

12
Getting Support on the Internet

In this chapter, we will talk about:

- Exploring online LaTeX resources
- Using web forums
- Framing a really good question
- Creating a minimal example

Introduction

In the early times of LaTeX, most support came from user groups, for example, installation disks and personal assistance. With the rise of the Internet, most information is just a fingertip away. There are still user groups such as TUG and DANTE behind the major resources, such as CTAN. But you can install updates and locate information very independently. Furthermore, you can get help from the TeX online communities. This chapter will show you how.

Exploring online LaTeX resources

There is a large number of LaTeX websites on the Internet. We have built a small guide with some very good starting points.

How to do it...

We will now look at a list of Internet addresses with a short description. Click on any of them and explore it. It leads to deeper information and to further sites.

Software archives and catalogues

These sites work as archives and catalogue sites that help browse the vast amount of software:

- `http://www.ctan.org` is the most relevant software and package archive. **CTAN** stands for **Comprehensive TeX Archive Network**. It provides about 5000 packages for TeX and LaTeX and related tools. You can browse packages by name and by topic, and use a site search. Most important is its hidden quality as a distribution network for TeX installations such as TeX Live and MiKTeX. It consists of a central server and mirrors servers across the whole world.
- `http://texcatalogue.ctan.org` is a maintained catalogue of most TeX and LaTeX packages that can be browsed by topic, alphabetically, or in a logical hierarchy
- `http://www.tug.dk/FontCatalogue/` is the LaTeX Font Catalogue, which offers a sorted index with examples of most fonts with direct LaTeX support

User groups

The long-established user groups are also the backbone of LaTeX on the the Internet. They stand behind CTAN, but also have their own home pages with a lot of information. The following web sites are their home pages:

- `http://www.tug.org` is the home of the international **TeX User Group** (**TUG**). Their website is like a portal to the TeX world.
- `http://uk.tug.org` is the website of the UK-TUG, the UK TeX User Group. They also offer LaTeX courses.
- `http://www.dante.de` is the site of the German language TeX users association, **DANTE e.V**. It has a lot of links to information in German.
- `http://www.latex-project.org` is actually more of a developer group. It belongs to the team that develops the next LaTeX version, and provides information updates.

Web forums and discussion groups

The sites with the most activity on the Internet are the web forums. Usenet discussion groups were the earliest discussion places, and they are still used today. These are some of the most relevant sites:

- `http://www.latex-community.org/` is primarily a very active and mature LaTeX support forum with more than 100,000 posts, categorized, tagged, and searchable. Questions are usually quickly answered. In addition, it maintains an article archive and a LaTeX news portal.
- `http://tex.stackexchange.com/` is a pure question and answer site, and is very active. There are no news updates or articles.
- `http://texwelt.de/` is a very active question and answer site in German.
- `http://www.golatex.de/` is another German web forum. It is well frequented and has a LaTeX wiki in addition.
- `comp.text.tex` is a Usenet discussion group for TeX and LaTeX, accessible using a Usenet reader or Google Groups.
- `de.comp.text.tex` is a German Usenet discussion group like the previous one.

Web forums can be accessed with any Internet browser, such as Firefox, Opera, Safari, or Internet Explorer. You can even read them on smart phones.

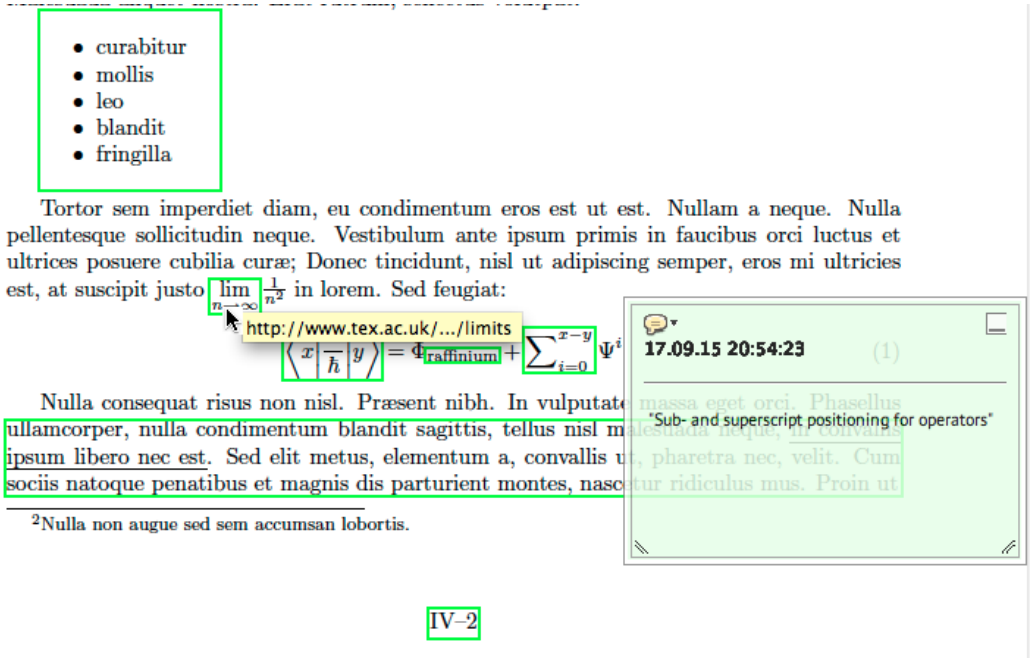
For Usenet groups, you need a Usenet reader software or can browse them using Google Groups as a web interface.

Frequently Asked Questions

Discussion groups, web forums, and mailing lists collect frequently recurring questions and put them online. If you have a question, first check there to see if it already has an answer. This can save time. Here is a list of sites with answers to **Frequently Asked Questions** (**FAQs**):

- `http://www.tex.ac.uk` is the English FAQs, collected by the UK-TUG. Most topics are covered with links to recommended packages.
- `http://www.ctan.org/tex-archive/info/l2picfaq` is a question and answer collection about working with images and floats.
- `http://tug.org/mactex/faq/` is the FAQ for MacTeX and Mac users.
- `http://projekte.dante.de/DanteFAQ` and `http://texfragen.de` are German FAQ sites.
- `http://texdoc.net/pkg/visualfaq` is a visual LaTeX FAQ. It provides a document with more than 100 text samples. Click on any detail marked in green and it will lead you to the corresponding page of the UK TeX FAQ. You will see the mouse cursor moving to an operator index; when a comment pops up and the link to the FAQ entry is shown. Now you are just one click away from a solution.

The following screenshot shows working with the visual FAQ:



There's more...

Besides my own blog, `http://texblog.net`, there are many TeX and LaTeX blogs. Most of them are featured in the Community aggregator on `http://texample.net`. On the front page, you can see the newest posts from the blog world. Blogs and post extracts are archived there.

If you would prefer e-mail LaTeX support, there are various mailing lists where you can subscribe. I have listed collections on `http://texblog.net/latex-link-archive/mailinglists/`.

Using web forums

The earliest online LaTeX support happened via Usenet and mailing lists. Today, we have Internet forums, which are very convenient to use.

Experienced users and experts gladly support LaTeX in forums. The reason for the existence of forums is not just expert chat. The majority of topics are based on users asking questions and readers answering them. Users' questions are the seed of forum life. So you could ask anything LaTeX related there and it would be very welcome. Let's see how to do this.

How to do it...

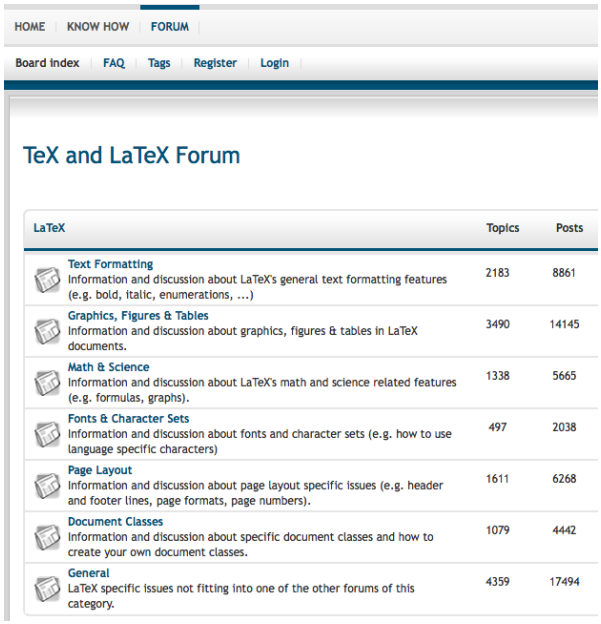
As an example, we will visit the LaTeX Community forum. It has existed since January 2008, and had about 100,000 posts at the time this book was published. I maintain this forum, and I can tell you this: *we love good questions!*

So let's have a walk together:

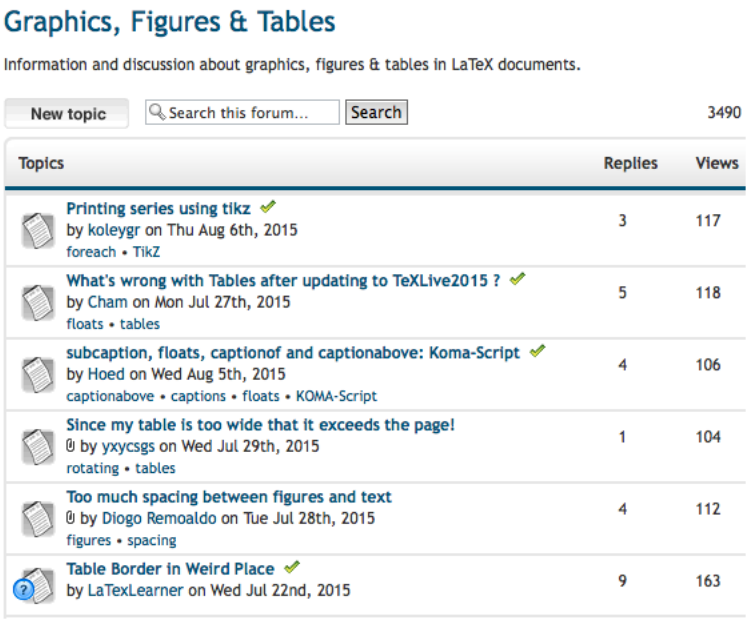
1. With any Internet browser, visit the address `http://latex-community.org`. The top of the browser window will show a menu as follows:



2. Click on **FORUM** in the top menu or on the left of the red banner. Alternatively, you can visit `http://latex-community.org/forum/` directly and bookmark it in your browser. You can now see the forum categories, such as LaTeX topics, editors, tools, distributions, and templates, in the following screenshot:



3. Click the header of any category, such as **Graphics, Figures & Tables**, to see the forum category view:



4. You can click the bold topic titles to browse around, use the search field in the top-right corner (seen in the first screenshot of this recipe), or click the **Tags** button at the top to see the tag cloud of about a 1,000 topics.
5. To start a new topic, such as for posting a new question, click on the button **New topic** in the top-left corner.

How it works...

While you can freely read everything without registering on the forum, posting needs registration. This is only because without this the forum gets spam posts from advertisers, causing much deletion work for moderators. So to write posts, please register with a login alias name you like. The advantage of this is that you can subscribe to topics and receive an e-mail if your topic gets an answer.

For writing posts, an editor offers the usual formatting tools such as these:

- Switching to bold or italic or to another font size or color
- Quoting parts of previous posts
- Formatting as numbered or bulleted lists

- Adding images
- Inserting hyperlinks (URLs)
- Adding attachments such as PDFs or log files

You also have LaTeX-specific features, such as these:

- A code button that changes your code snippets to human-readable code with LaTeX syntax highlighting, and a link that lets you open the code with a single click in an online LaTeX editor, which automatically compiles your code
- A button for inline LaTeX code within text
- A CTAN button that makes a package name a link to the CTAN package homepage
- A documentation button that makes a keyword a link to the corresponding manual on `TeXdoc.net`
- Tags for assigning keywords to topics and tag filtering
- Topic statuses such as "solved" so that we can filter for unsolved questions

The last-mentioned features in particular, and more shortcuts, allow users to work with LaTeX, CTAN, and documentation as easily as possible. This distinguishes the LaTeX forum from other, general forums or question and answer sites.

As I maintain the site, your question about LaTeX in that forum directly reaches me since I visit the forum daily and read through the new topics. I'm happy to provide support to all readers of this book, especially about the book's examples.

There's more...

As mentioned, another great site for getting help is the commercial question and answer site `http://tex.stackexchange.com`. I am a moderator here, and I've already explained this website in detail in the book *LaTeX Beginner's Guide*, which is also available from *Packt Publishing*.

Here are some further web forums for LaTeX:

- `http://texwelt.de` is a question and answer site for TeX and LaTeX in German
- `http://golatex.de` is a LaTeX web forum in German
- `http://texnique.fr` is a young question and answer site for TeX and LaTeX in French

These three forums are also maintained by me, so you can reach me there too.

Framing a really good question

As I mentioned previously, I love *good* questions, and most forum users do too. Sometimes, I will spend an hour creating a TikZ drawing for a user. Others work hard to troubleshoot error messages or output problems. We usually have fun with it, in the case of well-framed questions.

How to do it...

Challenges make us happy, especially if they are solvable. So, questions should do the following:

- Provide as much information as possible
- State the error message text, if there is one
- Format code actually as code using the syntax highlighting feature
- Show your own effort if you managed to achieve something

It's also recommended that you do a forum search with good keywords before posting. A similar question may already have been answered.

And there's the ultimate advice to do something which is nearly a guarantee to solve every LaTeX problem on earth: post a minimal example. The next recipe will deal with this.

Creating a minimal example

The best way to get help from someone is by handing the problem to them on a silver platter. They'll have no need to ask for further details and no need to guess, and the work won't be too boring. Describe the problem with a code example that meets these criteria:

- It should be *complete*, so we can try to compile it
- It should actually *show the problem*
- It should be *as small as possible*

This usually makes the problem easy to solve. It's handy enough for posting in a forum and the readers can easily try it out.

How to do it...

A good strategy is "divide and conquer". This refers to a technique of solving a problem by splitting it recursively into smaller problems until just simple problems remain. In our case, we repeatedly divide and simplify until we narrow it down, so we see the solution ourselves or cannot simplify it anymore, so it's ready to be shown to others.

Follow these steps to isolate the cause of a problem:

1. Create a copy of your document. If it consists of several files, copy them all. Continue only on this copy, not on the original.
2. Remove a significant part of the document that you presume doesn't contain the cause of the problem. You could do this by these means:
   - Moving the `\end{document}` command upwards
   - Deleting lines
   - Commenting out lines by placing a percent sign `%` in front
   - Commenting out or removing an `\include` or `\input` command
   - In the case of an included file, you can do this by deleting lines or commenting them out, or inserting the command `\endinput`, which you can also move upwards later on
3. Compile the document again.
   - If the problem persists and it's not narrowed down as much as possible, go again to step 2 and remove another part.
   - If the problem has gone, the cause must lie in the removed part. Restore that part, say using the editor's undo feature, and continue doing step 2, removing other parts of the document and reducing the affected part.
4. Simplify the document further, as follows:
   1. Remove non-relevant packages as you did in step 2 and 3 by commenting them out or by deleting the corresponding `\usepackage` line and compile the document again.
   2. Remove your own macro and environment definitions if they are not relevant.
   3. Remove images or replace them with a rectangle using the `\rule{...}{...}` command or give the option `demo` to the `graphicx` package as follows:

      ```
      \usepackage[demo]{graphicx}
      ```

   4. Replace long sections of text with generated dummy text, for example by using one of the packages `blindtext`, `lipsum`, or `kantlipsum`.
   5. Reduce long math formulas.
   6. Replace a bibliography file with the `filecontents*` environment of the `filecontents` package. You can find an example for this at `http://latex-cookbook.net/cookbook/examples/tag/bibliography`.
5. Check whether the document is now simple enough or if the preceding steps need to be repeated. The chances are high that your isolating effort already revealed the solution to you. Otherwise, the reduced example is ready to be posted in a forum.

Forum regulars commonly love code problems that they can copy and paste, fix, and verify the solution and post the answer to. If I see a good reduced example in a forum, I gladly feel obliged to test it and to try to solve the issue.

Don't worry that the procedure I detailed looks like much effort. It's just the thorough explanation that is a bit longish. Often just a few rounds of removing and testing is sufficient.

There's more...

In the preceding section, we used a top-down approach. This is a safe way in the sense that it contains the problem for sure.

Alternatively, we could take a bottom-up approach: begin with a small test document and build it up to show the problem. Then use this as the reduced example. If you roughly know where the cause of the problem is, it may be easier than narrowing it down in the whole copy of the original document. But the challenge is to reproduce the problem. If this would not happen, the reduced example would not be relevant.

With the bottom-down approach, we have the same goal but need to modify the requirements for our new setup. The document should meet these criteria:

- It should be *complete*, right from `\documentclass` down to `\end{document}`
- It should *show the problem* when compiled
- It should be *as small as possible*
- It should be usable on any basic LaTeX system

Ways to achieve the last point can be as follows:

- Use a standard class like `article`, `book`, or `report`
- Avoid system-dependent settings, such as input encoding and uncommon fonts
- Try loading only commonly used packages

Forum readers or other remote helpers usually don't like to install something just to test your code.

The `mwe` package is very handy for creating reduced examples. It automatically loads the `blindtext` and `graphicx` packages and provides several dummy images; this makes a minimal example elegant.

Let's have a short demonstration. Let's look at a short code snippet that shows how to wrap text around an image. We neither have an image file, nor do we want to type much text. We will follow these steps:

1. Load the `mwe` package

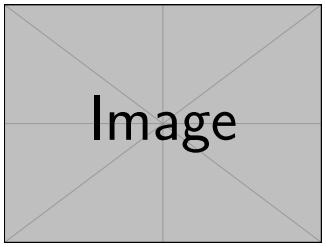
2. Use the `\blindtext` command for dummy text
3. As an image file name, just use the `example-image` command.

For demonstrating these steps, we will wrap some text around an image. Use the following commands:

```
\documentclass{article}
\usepackage{mwe}
\usepackage{wrapfig}
\begin{document}
\noindent
\blindtext
\begin{wrapfigure}{m}{5cm}
  \includegraphics[width=5cm]{example-image}
\end{wrapfigure}
\blindtext
\end{document}
```

When you compile these commands, you will get the desired images and text as follows:

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.



When class features are not relevant, a convenient way to show sample output is using the `standalone` class. It automatically crops the PDF to the actual content. We used this class in *Chapter 10*, *Advanced Mathematics*.

A reduced example showing the problem is almost a guarantee of a solution. Either the forum readers can fix it for you or you may even find the cause yourself in the reduction process.

Index

A

accented characters
  inputting 54, 55
acronyms
  list, creating 191, 192
algorithm
  typesetting 308-311
animation
  creating 220-222
atoms
  representing 332, 333

B

bar chart
  building 240-243
beamer
  package 171
  URL 32
bibliography
  creating 183-188
BibTeX 183
blend mode 249
blogs
  URL 342
bold mathematical symbols
  AMS-LaTeX and amsmath 101
  bm and amsmath, comparing 101, 102
  Standard LaTeX 101
  writing 99, 100
book
  cover page, adding 22
  designing 16-20
  document class, changing 23
  page layout, changing 20, 21
  title page, designing 21, 22
booktabs package 150-152
boxes of letters
  visualizing 66
boxes of symbols
  visualizing 66
bubble diagram 227

C

calculations 303-305
calendar
  producing 144-147
Cascading Style Sheet (CSS) 345
cells
  merging 162-165
  spanning, over multiple rows 165, 166
  splitting, diagonally 166-168
chemical formulae
  writing 322-325
chemistry
  package list, URL 325
circular diagram 226
ClassicThesis package 9
cloud chart 246
code listing
  printing 311-314
coloredlettrine package
  URL 77
colorful box
  text, inserting into 59-61
colortbl package 162
columns
  styles, adding to 173-175
Command Prompt
  URL 183

comments
fillable forms, producing 211-214
inserting 208-211
commutative diagrams
drawing 281-286
Comprehensive TeX Archive Network (CTAN)
URL 340
constellation diagram 228
contour
adding 110-112
copyright information
adding 206, 207
CTAN
about 166
package list, URL 325
URL 33
Curriculum Vitae (CV) 33-36
custom hyperlinks, inserting
Internet, linking to 201, 202
labeled objects, linking to 201
place linking to, within document 200, 201

D

DANTE e.V
URL 340
data
importing, from files 176-178
dcolumn package 159
decision tree
creating 238-240
definitions
stating 274-277
descriptive diagram 229
detexify tool
URL 260
Device independent file (DVI) format 197
dimensions
axes, reducing 294, 295
plotting 292-294
double stroke letters
writing, on blackboard 106, 107

E

EB Garamond font
about 77
OpenType version 77
URL 78
e-book readers
output, optimizing 215, 216
electrical circuits
drawing 334-337
extensions
bundle 2
document class 1
package 2
template 2

F

figures list
tuning 180
files
data, importing from 176-178
flowchart
constructing 231-236
fmtcount package
about 58
enumerated lists support 58, 59
gender support 58
multilingual support 58
fonts
attributes 88, 89
changing, commands 95
combinations 92, 93
environment, for changing 96, 97
Kepler fonts 92
Latin Modern 91
locally switching to 94, 95
selecting, for document 90, 91
single symbol, importing from font family 97-99
URL 87
footnotes
adding, to table 154-157

formula
highlighting in 270-273
forum, LaTeX
URL 54
frame
adding, to image 119, 120
Frequently Asked Question (FAQ)
for MacTeX, URL 341
URL 341
functions
axes, reducing 289, 290
plotting, in two dimensions 287, 288
polar coordinates, plotting in 290, 291
ticks and grid, adding 289

G

geometry pictures
drawing 295-298
labels, printing 300-302
objects, drawing 300
points, calculating 299
points, defining 298
glossary
adding 188-190
glue 67
graph theory
application 315-317
grid
typesetting 67-71

H

headings
preparing 142-144
hyperlinks
adding 198, 199
backreferences, getting in bibliography 203
color, changing 202
custom hyperlinks, inserting 200
of index entries 203
shape, changing 202
hyperref
URL 199
hyperxmp command
URL 207
hyphenation
improving 55, 56

I

iconv
URL 55
ImageMagick 86
images
aligning 126, 127
arranging, in grid 128, 129
background image, adding 133-138
cutting, for rounded corners 120-122
drawing over 124, 125
figure position, fixing 117
floating figures, limiting 117
frame, adding 119, 120
including, with optimal quality 114-116
manipulating 118
positioning, automating 116, 117
shaping, like circle 122, 123
stacking 130-132
tables, limiting 117
index
producing 193-196

J

JavaScript
URL 214
justification
improving 55, 56

K

kern 67
keys
mimicking 147, 148
KOMA-Script like sans-serif headings
getting 182, 183

L

l2tabu 8
large poster
creating 45-52
LaTeX
about 179
community, URL 341
editors 3
editors, URL 3

Font Catalogue, URL 340
forums, URL 343
online LaTeX resources 339
prerequisites 2
project, URL 340
LaTeX2e 8
LaTeX distribution 2, 3
latexmk command
URL 194
Launchpad
URL 124
layout
geometry, using 64
page layout details, examining 64, 65
visualizing 62-64
leaflet
background image, adding 44
fold marks and cut lines 44
margins, adjusting 44
producing 40-44
sectioning font, changing 45
legible table
creating 150-153
letter
enclosures, adding 39
paragraphs, separating 39
signature, changing 39
writing 37-39
ligatures
copying, enabling 108-110
search, enabling 108, 110
line-breaking
in equations 268, 269
LuaLaTeX 66, 112

M

MacTeX
URL 2
mailing lists
URL 342
math
basics 258
displayed math 259
greek letters 259
inline math 259
mode 259
numbering 263
referencing equations 263
squares and fractions 261
symbols 259, 260
math formulas
fine-tuning 266, 267
menu items 147, 148
metadata
adding 203-206
MiKTeX
about 166
URL 2
mind map 250-253
minimal example
creating 346-350
molecules
building blocks, using 330
drawing 326-328
naming 329
ready-drawn carbohydrates, using 331, 332
rings, drawing 328
TikZ options, applying 330
multi-line formulas
block of equations, centering 264
relation symbol, aligning at 264
writing, with alignment 263
multiple rows
cells, spanning over 165, 166

N

numbering
adjusting 265
numbers
converting, to words 57, 58
numeric data
aligning 157-159

O

online LaTeX resources
frequently Asked Questions 341, 342
software archives and catalogues 340
user groups 340
web forums and discussion groups 341
operators 262

ornaments
creating 138-142
URL 138
OT1 56
output
optimizing, for e-book readers 215, 216
over-sized letter
used, for starting paragraph 75, 76

P

paragraph
drop cap size, modifying 76
initial, coloring 77-79
starting, with over-sized letter 75, 76
PDF files
combining 219, 220
pdfLaTeX 197
pdfTeX 197
penalty 67
physical quantities
writing, with units 319-321
pie chart
cloud chart 246
drawing 243-245
polar area chart 246
square chart 245
polar area chart 246
Portable Document Format (PDF) 197
Portable Graphics Format (pgf) 172, 224
PostScript (PS) 197
preamble 8
presentation
color, changing 32
creating 24-29
font, changing 31
frames, splitting in columns 30
hand-out, providing 32
information piecewise, uncovering 30
navigation symbols, removing 31
outline, displaying for each section 31
short titles and names, using 29
theme, loading from Internet 32
priority descriptive diagram 230
proTeXt
URL 3
PSTricks 62, 224
pull quote
creating 82-86
pullquote.dtx file
URL 82

Q

questions
framing 346

R

rccol package 159
ready-drawn carbohydrates
using 331, 332
recode
URL 55
rows
styles, adding to 173-175
rubber space 69
rules 152

S

sans serif mathematics font
alternative approach 105
Arev Sans 105
getting 103, 104
Kepler fonts 106
with direct math support 105
screenread package
URL 217
script 4
shading
adding, to table 168-172
shape
adding, to table 168-172
commands 81
cutting out 81
text, fitting to 79, 80
short text
writing 4-8
single symbol
importing, from font family 97-99
siunitx package 157
skins 62
slashbox package 168

smart diagrams
animating 231
bubble diagram 227
building 225, 226
circular diagram 226
constellation diagram 228
descriptive diagram 229
priority descriptive diagram 230
square chart 245
styles
adding, to columns 173-175
adding, to rows 173-175
subscript 261, 262
superscripts 261, 262

T

table
coloring 160-162
footnotes, adding to 154-157
shading, adding to 168-172
shape, adding to 168-172
transparency, adding to 168-172
tuning 180
table notes
about 154
benefits 154
table of contents (TOC)
flat, printing 181, 182
KOMA-Script like sans-serif
headings 182, 183
tuning 180
widths, automatic correction 180
table rows
numbering, automatically 172
tabular* environment 149
tabularx environment 149
tabulary environment 149
tcolorbox package 62
Terminal 217, 218
TeX
URL 340
TeX distribution 2, 3
texdoc
URL 298
texdoc algorithmicx command
URL 311
texdoc amsmath command
URL 305
texdoc amsthm command
URL 277
texdoc animate command
URL 222
texdoc babel command
about 56
URL 56
texdoc background command
URL 138
texdoc biblatex command
URL 188
texdoc bohr command
URL 334
texdoc chemformula command
URL 325
texdoc command
URL 75
texdoc datatool
URL 178
texdoc fontenc command
URL 56
texdoc graphicx command
URL 118
texdoc inputenc
URL 14
texdoc lettrine command
about 79
URL 79
texdoc mathmode command
URL 305
texdoc memdesign command
URL 23
texdoc menukeys command
URL 148
texdoc multirow command
URL 166
texdoc ntheorem command
URL 278
texdoc pgfplots in command
URL 243
texdoc scrguien command
URL 21
texdoc selinput command
about 55
URL 55

texdoc siunitx command
URL 321
texdoc stackengine command
URL 132
texdoc symbols command
URL 260
texdoc tcolorbox command
URL 62
texdoc threeparttable command
URL 157
texdoc tikz command
URL 144
texdoc tocstyle command
URL 183
texdoc xcolor command
URL 77
TeX installation
URL 224
TeX Live
about 166
URL 2
URL, for downloading 166
text
absolute positioning 71-75
fitting, to shape 79, 80
inserting, into colorful box 59-61
TeX User Group (TUG)
URL 340
TeXworks 3
theorems
alternative theorem package 278
numbering, adjusting 278
stating 274-277
tools 279, 280
thesis
displayed equations, centering 16
input encoding, changing 14
layout of captions, modifying 15
margins, changing 15
table of contents, getting 15
writing 8-14
threeparttable package 154
TikZ
about 62, 86, 171
example gallery, URL 305
gallery, URL 147
manual, URL 337
options, applying 330
timeline
generating 253-256
tocstyle 183
transparency
adding, to table 168-172
tree
building 236, 237
decision tree, creating 238, 239
tufte-latex class
URL 23
typographers
advice 153

U

UK-TUG
URL 340
units
used, for writing physical quantities 319-321

V

Venn diagram
drawing 247-249
visual LaTeX FAQ
URL 341

W

web forums
URL 345
using 342-345
white margins
removing 217, 218
words
numbers, converting to 57, 58
WYSIWYG (What You See Is What You Get) 150

X

xcolor package 160, 162
XeLaTeX 112



Thank you for buying
LaTeX Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

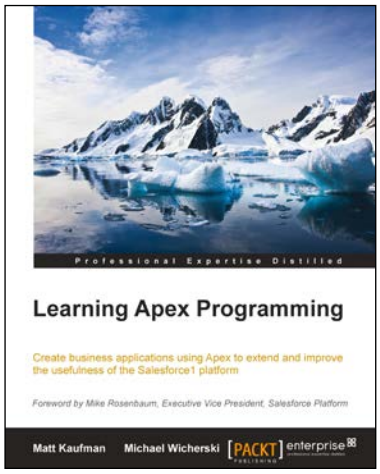
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt open source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's open source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

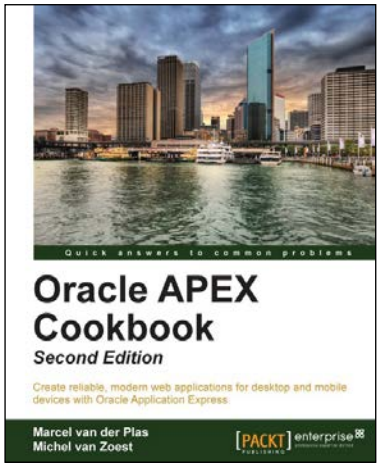


Learning Apex Programming

ISBN: 978-1-78217-397-7 Paperback: 302 pages

Create business applications using Apex to extend and improve the usefulness of the Salesforce1 Platform

1. Create Apex triggers and classes and build interactive Visualforce pages.
2. Understand best practices and workarounds to platform limitations.
3. Hands-on examples that will help you create business applications using Apex quickly and efficiently.



Oracle APEX Cookbook
Second Edition

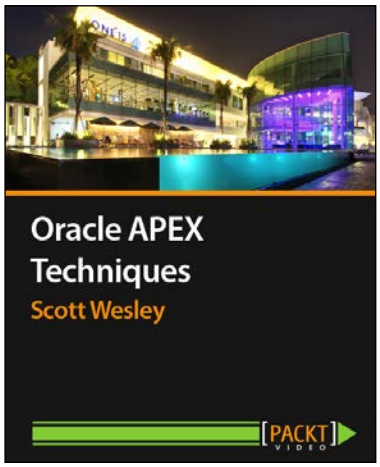
ISBN: 978-1-78217-967-2 Paperback: 444 pages

Create reliable, modern web applications for desktop and mobile devices with Oracle Application Express

1. Explore APEX to build applications with the latest techniques in AJAX and Javascript using features such as plugins and dynamic actions.
2. With HTML5 and CSS3 support, make the most out of the possibilities that APEX has to offer.
3. Part of Packt's Cookbook series: Each recipe is a carefully organized sequence of instructions to complete the task as efficiently as possible.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

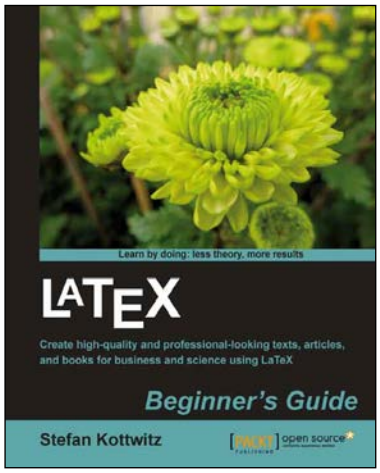


Oracle APEX Techniques [Video]

ISBN: 978-1-84968-934-2 Duration: 02:00 hrs

Go beyond the basics and explore how to make the most of the Oracle Application Express environment

1. Make the most of the extensibility of Oracle APEX 4.2.
2. Deploy dynamic actions to make your application more interactive without needing to learn JavaScript.
3. Provided with practical examples and solutions to problems you will encounter as you start to push the product to fit your requirements.



LaTeX Beginner's Guide

ISBN: 978-1-84719-986-7 Paperback: 336 pages

Create high-quality and professional-looking texts, articles, and books for business and science using Latex

1. Use LaTeX's powerful features to produce professionally designed texts.
2. Install LaTeX; download, set up, and use additional styles, templates, and tools.
3. Typeset math formulas and scientific expressions to the highest standards.
4. Include graphics and work with figures and tables.