

```

4|<!ELEMENT node (name,WAShome,WASversion,profile,
|hostname+,ipaddr,server+)>
5|<!ELEMENT server (name,endpoint+)>
6|<!ELEMENT name (#PCDATA)>
7|<!ELEMENT WAShome (#PCDATA)>
8|<!ELEMENT WASversion (#PCDATA)>
9|<!ELEMENT profile (#PCDATA)>
10|<!ELEMENT hostname (#PCDATA)>
11|<!ELEMENT ipaddr (#PCDATA)>
12|<!ELEMENT endpoint (name,port)>
13|<!ELEMENT port (#PCDATA)>

```

Table 22-3 describes the DTD in Figure 22-11, line by line. Using the description and the WASports_10.py application, you should be able to better understand the correlation between the application output and the DTD format.

¹³Note that line 4 is too long to fit in the available space and is continued on the next line. ¹⁴For example, see http://www.w3schools.com/dtd/dtd_intro.asp.

Table 22-3. WASports.dtd, *Explained*

Lines Description

- 1 The root tag is WASports and it has only one cell tag.
- 2 The WASports tag has a required attribute called version.
- 3 The cell tag requires a name, WAShome, WASversion, profile, and one or more node tags.
- 4 The node tag requires a name, WAShome, WASversion, profile, one or more hostnames, one ip_addr and one or more server tags.
- 5 Each server tag requires one name and one or more endpoint tags.
- 12 Each endpoint tag requires a name and port tag.
- 6–13 The tags identified as having (#PCDATA) require some character data.

The ExportTask Class

You need to make some changes to the application menu structure, in order to provide the user with a way to specify that an export should occur. Creating the menu is simple enough that it doesn't need to be shown here. However, the event handler that is called requires a bit more explanation, so the Export(...) method is

shown in Listing 22-36.

Listing 22-36. Export(...) Method from WASports_10.py

```
788| def Export( self, event ) :
789| title = 'Export (Save) cell details'
790| fc = JFileChooser(
791| currentDirectory = File( '.' ),
792| dialogTitle = title,
793| fileFilter = XMLfiles()
794| )
795| if fc.showOpenDialog(
796| self.frame
797| ) == JFileChooser.APPROVE_OPTION :
798| f = fc.getSelectedFile()
799| fileName = fc.getSelectedFile().getAbsolutePath()
800| if not fileName.endswith( '.xml' ) :
801| fileName += '.xml'
802| msg = 'Overwrite existing file (%s)?'
803| if os.path.isfile( fileName ) :
804| response = JOptionPane.showConfirmDialog(
805| None,
806| msg % os.path.basename( fileName ),
807| 'Confirm Overwrite',
808| JOptionPane.OK_CANCEL_OPTION,
809| JOptionPane.QUESTION_MESSAGE
810| )
811| if response == JOptionPane.CANCEL_OPTION :
812| return
813| ExportTask( fileName, self.cellData ).execute()
```

This method uses the JFileChooser and FileFilter classes discussed in detail in Chapter 17. It limits the kinds of files displayed by the JFileChooser to those having an .xml extension. If the user selects an existing file, it also verifies that the user wants to replace the existing file using a simple confirmation dialog window. The most important part of this method is when it instantiates and begins execution of an ExportTask thread to actually create the specified file (line 813). Listing 22-37 shows the first of three parts of the ExportTask class.

Listing 22-37. Part 1 of the ExportTask Class from WASports_10.py

```

356|class ExportTask( SwingWorker ) :
357| def __init__( self, fileName, cellData ) :
358| self.fileName = fileName
359| self.cellData = cellData
360| def doInBackground( self ) :
361| try :
362| fos = FileOutputStream( self.fileName )
363| streamResult = StreamResult(
364| OutputStreamWriter( fos, 'ISO-8859-1' )
365| )
366| trans = SAXTransformerFactory.newInstance()
367| trans.setAttribute( 'indent-number', 4 )
368| tHand = trans.newTransformerHandler()
369| serializer = tHand.getTransformer()
370| serializer.setOutputProperty(
371| OutputKeys.ENCODING, 'ISO-8859-1'
372| )
373| serializer.setOutputProperty(
374| OutputKeys.DOCTYPE_SYSTEM, 'WASports.dtd'
375| )
376| serializer.setOutputProperty(
377| OutputKeys.INDENT, 'yes'
378| )
379| tHand.setResult( streamResult )
380| tHand.startDocument()
381| atts = AttributesImpl()
382| atts.addAttribute(
383| " ", " ", 'version', 'CDATA', __version__
384| )
385| tHand.startElement( " ", " ", 'WASports', atts )
386| atts.clear()
387| tHand.startElement( " ", " ", 'cell', atts )
388| data = self.cellData
389| root = data.tree.getModel().getRoot()
390| cellName = root.toString()
391| self.addTagAndText( tHand, 'name', cellName )
392| info = data.getInfoDict( cellName )
393| self.addTagAndText(
394| tHand, 'WAShome', info[ 'WAShome' ]

```

```

395| )
396| self.addTagAndText(
397| tHand, 'WASversion', info[ 'WASversion' ]
398| )
399| self.addTagAndText(
400| tHand, 'profile', info[ 'profile' ] 401| )

```

Listing 22-38 shows the next part of the ExportTask class. Again, you can see that a non-trivial class can't easily be forced to fit onto one or even two pages of this book.

Listing 22-38. Part 2 of the ExportTask Class from WASports_10.py

```

402| nodes = root.children()
403| while nodes.hasMoreElements() :
404| node = nodes.nextElement()
405| nodeName = node.toString()
406| info = data.getInfoDict( nodeName )
407| tHand.startElement( ", ", 'node', atts )
408| self.addTagAndText( tHand, 'name', nodeName )
409| self.addTagAndText(
410| tHand, 'WAShome', info[ 'WAShome' ]
411| )
412| self.addTagAndText(
413| tHand, 'WASversion', info[ 'WASversion' ]
414| )
415| self.addTagAndText(
416| tHand, 'profile', info[ 'profile' ]
417| )
418| self.addTagAndText(
419| tHand, 'hostname', info[ 'hostnames' ]
420| )
421| self.addTagAndText(
422| tHand, 'ip_addr', info[ 'ipaddr' ]
423| )
424| servers = node.children()
425| while servers.hasMoreElements() :
426| server = servers.nextElement()
427| serverName = server.toString()
428| tHand.startElement(

```

```

429| ", ", 'server', atts
430| )
431| self.addTagAndText(
432| tHand, 'name', serverName
433| )
434| table = data.getPortTable(
435| ( nodeName, serverName )
436| )
437| model = table.getModel()
438| for row in range( model.getRowCount() ) :
439| tHand.startElement(
440| ", ", 'endpoint', atts
441| )
442| name = model.getValueAt( row, 1 ) Listing 22-39 shows the remainder of
the ExportTask class. Unfortunately, breaking a listing like this across multiple
pages can make it difficult to read. Therefore, you are encouraged to use your
favorite text editor and view the script source file.

```

Listing 22-39. Part 3 of the ExportTask Class from WASports_10.py

```

443| self.addTagAndText(
444| tHand, 'name', name
445| )
446| port = str(
447| model.getValueAt( row, 0 )
448| )
449| self.addTagAndText(
450| tHand, 'port', port
451| )
452| tHand.endElement(
453| ", ", 'endpoint'
454| )
455| tHand.endElement( ", ", 'server' )
456| tHand.endElement( ", ", 'node' )
457| tHand.endElement( ", ", 'cell' )
458| tHand.endElement( ", ", 'WASports' )
459| tHand.endDocument()
460| except :
461| msgText = "\nExportTask() Error: %s\nvalue: %s"

```

```

462| print msgText % sys.exc_info()[ :2 ]
463| def addTagAndText( self, handler, tagName, text ) :
464| handler.startElement(
465| " ", tagName, AttributesImpl()
466| )
467| handler.characters(
468| String( text ).toCharArray(), 0, len( text )
469| )
470| handler.endElement( " ", tagName )

```

The ExportTask shown in Listings 22-37 through 22-39 is described in detail in Table 22-4. The important point to note about the ExportTask class is that every XML tag calls the startElement(...) method, the required nested elements, and the endElement(...) method in order to create a properly formed XML document.

Table 22-4. *ExportTask Class, Explained*
Lines Description

357–359 Class constructor used to save the specified filename and cellInfo structure containing the information to be exported.

362–365 The output file stream is instantiated.

366–367 A SAXTransformerFactory instance is created with the specified indentation (for readability).

368–379 A transformer handler instance is created with the specified properties (such as encoding). It is also associated with the output stream created earlier.

380 The startDocument(...) method begins to create the XML document being produced.

381–385 The WASports tag is created with the required version attribute.

386 Since no other tags in the document have attributes, the atts variable is cleared so it can be reused on other tags.

387 The only instance of the cell tag is started.

388–401 The tags required to be present in the cell tag hierarchy are created using the addTagAndText(...) utility method.

402–456 Each node in the cell has its required tags and all of the associated server entries are created.

457–459 Note how each startElement has a matching endElement after all the contained tags have been created.

460–462 An exception clause is invoked when an error is encountered. Unfortunately, it is a trivial error handler.

463–470 The addTagAndText(...) utility method simplifies the creation of tags like <tagName>text value</tagName>.

Step 11: Implementing the Import Functionality

One of the most challenging aspects of the iterative development of non-trivial applications such as WASports is that you can reach a point where significant changes are required for the next iteration. Step 11 is where these changes are recognized. How do I know that? Let's take some time to consider the implications associated with being able to import application information. To make it more interesting, you'll do so by using a few question and answers to get things started.

Table 22-5. Important Questions and Answers

Q: What should occur when a WASports file is imported?

A: A new internal frame should be created similar in nature to the one that is created when the application starts. Q: How will an imported frame differ from an existing one?

A: These frames can be closed, and if changes are made, a save needs to be performed using the export functionality, instead of changing the current configuration.

Q: Do changes to existing classes and data structures need to be made?

A: Yes; for example, the internal cellInfo class needs to be made global so that both local cell data and imported cell data can use the same class. Additionally, the setColumnWidths(...) method should be moved from within the PortLookupTask class to the global scope so that it can also be called by the ImportTask class. Additionally, the names dictionary in the cellInfo class can't refer to an actual configuration ID because you don't want any of the changes made to imported data to be made in the local configuration. So, for the imported data, a pseudoconfigId will be created instead.

Q: What kinds of things needed to be "fixed" because of improved understanding and how should they be handled? A: During this iteration, it became clear that the internal form of the JOption dialog boxes should have been

used instead of the default external form (that is, the `JOptionPane.showInternalConfirmDialog(...)` method instead of the `JOptionPane.showConfirmDialog(...)` method). Additionally, using an `ip_addr` tag for the `ipaddr` entry in the `cellInfo` class added unnecessary complexity. So the DTD and code was changed to use an `ipaddr` tag instead.

The ImportTask Class

The changes required to add the Import functionality are similar to the changes needed to add the Export functionality discussed previously. The `Import(...)` event handler method¹⁵ isn't shown here, but can be found in the `WASports_11.py` script file.¹⁶ It too uses a `JFileChooser` instance to allow the user to identify the XML file to be imported. Once the input file is identified, an `ImportTask` instance is created to perform the input processing on a separate thread. Listing 22-40 shows the `ImportTask` class used by this iteration of the `WASports` script.

Listing 22-40. Part 1 of the `ImportTask` Class from `WASports_11.py`

```
780|class ImportTask( SwingWorker ) :  
| ...  
992| def __init__( self, app, menuItem, fileName ) :  
993| self.app = app  
994| self.menuItem = menuItem  
995| self.fileName = fileName  
996| self.handler = ImportTask.SAXhandler( app )  
997| menuItem.setEnabled( 0 )  
998| self.msgText = "  
999| def doInBackground( self ) :  
1000| try :  
1001| FIS = FileInputStream( File( self.fileName ) ) 1002| ISR =  
InputStreamReader( FIS, ENCODING ) 1003| src = InputSource( ISR, encoding  
= ENCODING ) 1004| factory = SAXParserFactory.newInstance() 1005|  
factory.setValidating( 1 )  
1006| parser = factory.newSAXParser()  
1007| parser.parse( src, self.handler )  
1008| FIS.close()  
1009| except :  
1010| msgText = 'Error: %s\nvalue: %s'
```



```

1011| self.msgText = msgText % sys.exc_info()[ :2 ] 1012| def done( self ) :
1013| localFrame = self.app.localFrame
1014| desktop = self.app.frame.getContentPane() 1015| errors =
self.handler.errors
1016| if self.msgText or errors :
1017| if self.msgText :
1018| msg = self.msgText + '\n'
1019| else :
1020| msg = "
1021| for error in self.handler.errors :
1022| msg += ( '\n' + error )
1023| if msg.startswith( '\n' ) :
1024| msg = msg[ 1: ]
1025| JOptionPane.showInternalMessageDialog( 1026| desktop,
1027| msg,
1028| 'Import failed',
1029| JOptionPane.ERROR_MESSAGE,
1030| None
1031| )

```

¹⁵Note the use of capitalization to differentiate this method name from the import keyword. ¹⁶The complete source can be found in [...\code\Chap_22\WASports_11.py](#).

Listing 22-41 continues the ImportTask class started in the previous listing.

Listing 22-41. Part 2 of the ImportTask Class from WASports_11.py

```

1032| else :
1033| cellData = self.handler.getResults()
1034| tree = cellData.getTree()
1035| root = tree.getModel().getRoot()
1036| name = root.toString()
1037| if localFrame :
1038| size = localFrame.getSize()
1039| w, h = size.width, size.height
1040| count = len( self.app.frames )
1041| num = ( count - 1 ) % 8
1042| loc = Point(
1043| num * 27 + 32,
1044| num * 27 + 32

```

```

1045| )
1046| else :
1047| print '\nWarning: no localFrame' 1048| size = self.app.frame.getSize()
1049| w, h = size.width >> 1, size.height >> 1 1050| count = 0
1051| loc = Point( 32, 32 )
1052| internal = InternalFrame(
1053| title = '%d : %s' % ( count, name ), 1054| size = Dimension( w , h ),
1055| location = loc,
1056| cellData = cellData,
1057| app = self.app,
1058| closable = 1
1059| )
1060| self.app.frames.append(
1061| ( internal, self.fileName )
1062| )
1063| desktop.add( internal, None, 0 )
1064| internal.setSelected( 1 )
1065| node = tree.getModel().getRoot()
1066| tree.expandPath( TreePath( node.getPath() ) ) 1067|
self.menuItem.setEnabled( 1 )

```

It is interesting to see how much of this code deals with potential problems. This is one of the temptations that exist with rapid prototyping and the iterative development of applications. There is a tendency to leave out error checking and diagnostic or informational messages.

Unfortunately, this choice has its consequences, which are, by definition, self-inflicted and often painful. When you don't include error checking and diagnostic messages as you're developing your application, it's much more difficult to determine the source of any problems that crop up.¹⁷

Most of the code in Listings 22-40 and 22-41 should look familiar. For example, compare the `doInBackground(...)` method (lines 1001-1008) with the code in Listing 22-27. Both prepare the input stream using the appropriate encoding and instantiate a validating `SAXParserFactory` instance. They also both use this parser to process the specified input stream using a `SAXhandler` instance that has been customized for the application.

In this case, the `SAXhandler` class isn't shown, but is very similar, at least in

structure, to the one in Listing 22-33. The biggest difference is that instead of displaying the buffered character strings, they are saved in a `cellInfo` dictionary indexed by the associated tag name (the cell name is in an entry indexed by `cellName`).

The code is simplified by the fact that the parser validates the input against the DTD and calls one of the error-related routines if an error condition is detected (the `warning(...)`, `error(...)`, or `fatalError(...)` methods).

At the appropriate points in the parsing process (when the node `startElement` event is detected), all of the details required by the preceding hierarchy level (in this case, between the cell tag and the node tag) have been saved and can now be used to add a new tree element.

When the end of the XML document is encountered and no error conditions have been detected, the `done(...)` method process shown in Listing 22-40 uses the result of the processing to create a new inner frame instance and display it on the application desktop.

¹⁷Personal note: I learned the hard way that `SwingWorker` threads can fail silently (exceptions aren't displayed), so it is good to consider using `try/except` blocks in these kinds of threads.

What's Left?

At this point, the script is a realistic demonstration of the kinds of things that you can do to create a reasonable graphical interactive `wsadmin` application. Of course, there are features that you could add to make the application even more useful. The next sections discuss what you would need to do to add functionality to the script.

Text Highlighting

What would it take to add text highlighting to the application? It depends on what you want to highlight. For example, if you want to highlight the port table rows that contain specific text or specific port number values, you need to use a `TableCellRenderer` (see Chapter 12) to control how each cell should be rendered. Be careful, though; it is easy to forget that the `isSelected` argument of the `getTableCellRendererComponent(...)` method should be involved with determining how the table cell should be rendered.

Another possibility is to allow the user to highlight those table rows for active ports. However, remember to take into account the hostname or IP address of the local machine when determining whether imported port table rows should be highlighted.

There is also the possibility of adding multiple or combined highlighting conditions at the same time (both active and highlighted because of matching endpoint names or port numbers).

Maybe you want to allow the user to dynamically select the colors to be used for highlighting. If so, how would you allow the user to save this kind of preference information?

Table Sorting

It would be somewhat helpful to be able to sort and reorder the port tables. It isn't too difficult to implement,¹⁸ but the potential value is more difficult to judge. This is true because it is hard to guess how valuable being able to reorder the data will be to the users of your application.

Comparing Configurations

One interesting possibility to consider is related to the prospect of using this kind of application to compare two (or possibly more) configurations. What kinds of comparisons would you like to make? You could use an application like this to identify port conflicts and differences.

Wouldn't it be useful if the application could export the current port details and then import the previous configuration details and highlight the differences? Would you want it to be able to copy the port numbers from an imported XML file and use this information to change the values used in the local configuration? This kind of thing might be very useful if you wanted the environments to use identical port numbers.

Report Generation

Another potentially useful feature would be the capability to generate reports about the port numbers being used in one or more WebSphere environments. How might you want these reports organized? How important would it be for the data in these reports to be aggregates or filtering of the available data?

¹⁸See

<http://docs.oracle.com/javase/tutorial/uiswing/components/table.html#sorting>.

Summary

What next step makes the most sense to you? It depends on your specific needs. Can you use this application as it is? Maybe, maybe not; it depends on your needs. Consider using an application like this to view and manage all of the port numbers being used by your environment. Which would you prefer—to use an application like this or to use the WebSphere administration console to identify and manage the port numbers in your environment?

I think that you'll agree that this kind of application has real potential for making the administration of your WebSphere Application Server environment much simpler.

I hope that this book helps you on your journey of creating useful graphical wsadmin scripting applications.

Index



about(...) method, [443](#)
aboutTask.getResult() method, [443](#)
Absolute Layout Manager

advantages, [36](#)
application frame, [36](#)
frame.add(), [36](#)
getPreferredSize() method, [36](#)
repaint() method, [35](#)
setBounds() method, [36](#)

Absolute layout technique, [222](#)
AbstractAction class, [203](#)
AbstractCellEditor class, [178](#)
Accelerator, [132](#)
ActionEvent, [205](#)
ActionListener class, [126](#)

- ActionListener event handler, [31](#), [224](#)
- ActionListener method, [214](#)
- actionPerformed() method, [30–31](#), [47](#)
- Adapter classes, [209](#)
- addActionListener() method, [32](#)
- addButtons() method, [42–43](#)
- addCards() method, [43](#)
- addComponents() method, [61](#)
- add() method, [18](#)
- addTab() method, [51](#)
- AdminConfig methods, [423](#)
- AdminConfig.modify(...) method, [445](#)
- Administration console, [385](#), [419](#)
- AdminTask commands, [387](#)
- AdminTask.generateSecConfigReport() method, [416](#) AdminTask.help(...) method, [379](#)
- AdminTask.listServerPorts() method, [420](#)
- AdminTask methods, [420](#)
- AdminTask.reportConfiguredPorts() method, [422](#) appCleanup(...) routine, [444](#)
- Application Programming Interface (API) DOM, [448](#)
- XML (SAX), [449](#)

AutoResizeMode, [188](#)



Bindings

- AbstractAction class, [203](#)
- boundary conditions, [203](#)
- definition, [193](#)
- inputMaps and actionMaps

- getters and setters, [195](#)

- JComponent map attributes, [194](#) JTable hierarchy, [196](#)

- JTable map methods, [194](#)

- Boolean cell editors, [172](#)

- BorderLayout Manager advantages, [39](#)

application, [38](#)

BorderLayoutGap.py script, [40](#)

BorderLayoutNEWS.py script, [40](#)

component separation, [40](#)

directional constants, [38–39](#)

disadvantages, [39](#)

horizontal and vertical gaps identification, [41](#)

Box class strut methods, [56](#)

Box.createHorizontalBox() method, [54](#)

BoxLayoutDemo sample application, [51](#) BoxLayout Manager

Box class, [53](#)

boxes and resizable components, [57](#) BoxLayoutDemo class, [52–53](#)

BoxLayoutDemo sample application, [51](#) constants, [53](#)

constructor, [54](#)

createHorizontalBox() method, [54](#)

createVerticalBox() method, [54](#)

invisible box components

BoxLayout Manager (*cont.*)

glue, [55](#)

rigid area, [57](#)

strut, [56](#)

tabs, [51–52](#)

Button

ActionListener class, [30–31](#)

ActionListener event handler, [31](#)

actionPerformed() method, [31](#)

addActionListener() method, [32](#)

frame.add() method, [33](#)

java.awt.event.ActionListener classes, [30](#)

java.lang.Runnable class, [30](#)

multiple inheritance, [30](#)

trivial Java application, [29](#)

button() method, [47](#)

buttonPress() event-handling method, [43](#)



CardLayout Manager

`addButtons()` method, [42](#)

`addCards()` method, [43](#)

application, [41–42](#)

`buttonPress()` event-handling method, [43](#) `buttonPress()` method, [42–43](#)

`event.getActionCommand()` method, [43](#) `JPanel` instances, [42](#)

Layout Manager's `show()` method, [43](#)

`run()` method, [42](#)

Cell editors

Boolean cell editors, [172](#)

custom numeric cell editors, [174](#)

`JComboBox` cell editors, [175](#)

numeric cell editors, [173](#)

`cellInfo` `addOriginal(...)` method, [445](#)

Cell renderers

column-specific cell renderer, [166](#)

custom cell renderers, [163](#)

custom renderer

Boolean values, [165](#)

`TableSelection` output, [164](#)

data type rendering, [163](#)

data type-specific cell renderers, [162](#)

`DefaultTableCellRenderer` class, [165](#), [169](#) `getTableCellRendererComponent(...)`

method, [165](#) `setCellRenderer(...)` method, [166](#)

`TableCellRenderer` interface, [165](#)

`TableColumnModel`, [166](#)

`cellTree()` method, [427](#), [430](#)

`cellTSL` class, [428](#)

`ChangeListener` method, [363](#)

`characters(...)` method, [451](#)

class `run()` method, [13](#)

Column manipulation, [183](#)

column adjustments, [188](#)

column widths, [183](#)
auto resize mode, [189](#)
column heading width, [184](#) determination, [185](#)
getValueAt(...) method, [186](#) setColumnWidths(...) method, [186](#)
ComponentEventDemo.java, [217](#)
componentResized() method, [221](#)
Components, [1](#)
consoleTimeout script
GUI Jython version
AdminConfig methods, [339](#) output, [339](#)
run() method, [337](#)
SwingWorker class, [339–340](#) update () method, [338](#)
non-GUI Jython version, [336](#)
Container add() method, [23](#)
createEmptyBorder(...) method, [300](#) Cursor class
isSelected() method, [298](#)
WaitCursor1.py script, [297–298](#) CustomDialog class, [273](#)
Custom numeric cell editors, [174](#)



DefaultCellEditor, [174–175](#), [179–180](#)
DefaultTableCellRenderer, [162](#)
DefaultTableModel, [162](#)
Dialog boxes

CustomDialog class, [273](#)
definition, [263](#)
GraphicsConfiguration

component, [265](#)
JOptionPane methods, [274](#)
multiple modal dialog boxes, [273](#)
SimpleDialog class, [270](#)

Document Object Model (DOM), [448](#)
Document Type Definition (DTD), [452](#) doInBackground(...) method, [302](#), [305](#),
[439](#), [460](#) done(...) method, [439](#)



endElement(...) method, [451](#), [456](#)
event.getActionCommand() method, [43](#), [59](#) Event handler method, [443](#)
expandtabs() method, [355](#)
Export(...) method, [453](#)



FileFilter class, [454](#)
findScopedTypes(...) routine, [434](#) firstNamedConfigType(...) routine, [436](#)
FlowLayout Manager, [37](#)
FormattingVisitor() method, [248](#) frame.add() method, [23](#)
frame.getBounds() method, [221](#) Frame resize method, [400](#)



generateSecConfigReport, [388](#)
getAttributeValue(...) routine, [434](#)
getContentPane() method, [18](#), [300](#)
getFontMetrics(...) method, [371](#)
getHeaders class, [246](#)
getHostnames(...) routine, [434](#)
getIPAddresses(...) routine, [434](#)
getMonths() method, [93](#)
getPortTable(...) method, [437](#)
getPreferredSize() method, [36](#)
getTableCellEditorComponent(...) method, [171](#)
getTableCellRendererComponent(...) method, [184](#), [372](#) getViewport().getView()
methods, [363](#)
getWeekdays() method, [93](#)
Global security application

content pane, [17](#)
glass pane, [16–17](#)
JLabel, [15](#)
layered pane, [17](#)
optional MenuBar, [17](#)

Glue components, [56](#)

GraphicsConfiguration object, [267](#)
ScreenLoc class, [268](#)
ScreenPos class, [268](#)
GraphicsEnvironment class, [266](#)
GraphicsEnvironment session, [266](#)
GridLayout Manager
addButtons() method, [59](#)
addComponents() method, [61](#)
application, [58](#), [60](#)
buttonPress() method, [59](#)
displayConstraints() function, [61](#)
event.getActionCommand() method, [59](#)
GridBagConstraints class, [61–62](#)
layoutContainer() method, [60](#)
pane.getLayout() method, [60](#)
run() method, [59](#)
GroupLayout Manager, [62](#)



headerTask class, [247](#), [256](#)
Help.help() method, [355](#)
Help.wsadmin() method, [360](#)
Hypertext markup language (HTML)

FormattingVisitor() method, [248](#) getHeaders class, [246](#)
getHeader scripts, [249](#)
getLinks routine, [232](#)
headerTask class, [247](#)
head(...) method, [248](#)
HTML label, [243](#)
HTML text modification, [244](#) javadocInfo_03 sample output, [241](#) Java “HTML”
classes, [231](#)
JToggleButton, [245](#)
rendering HTML, [242](#)
textTask class, [241](#)



- IBM website, [233](#)
- Import(...) event handler method, [458](#)
- ImportTask class, [458](#)
- Inner cellInfo class, [436](#)
- InputMethodListener class, [114–115](#)
- InputVerifier method, [227](#)
- Interactive scripts
 - deprecation message, [11](#)
 - equivalent Java application, [10](#)
 - Welcome.py script file, [9](#)
- Internal frames
 - iFrameDemo class, [317–319](#)
 - JDesktopPane class, [322–324](#)
 - JInternalFrame classes (*see* JInternalFrame classes) layers
 - labels, [319](#)
 - LayeredPaneDemo class, [320–321](#)
 - positioning, [321–322](#)
 - scratch
 - application output, [347](#)
 - consoleTimeout script (*see* consoleTimeout script) InternalFrame class, [343–344](#)
 - menu items, [341–342](#)
 - multiple inheritance issue, [344](#)
 - RadioButton class, [349–351](#)
 - result, [352](#)
 - revised consoleTimeout class, [342](#)
 - setSelected() method, [342](#)
 - setValue(...) method, [350–351](#)
- Internal frames (*cont.*)
 - stateChange(...) method, [348](#)
 - TextandButton class, [347–348](#)
 - TextField class, [344–345](#)
 - wsadmin script, [336](#)
- WSAStask class, [346](#)
- Introspection, [4](#)
- invokeLater() method, [12](#)
- isCellEditable method, [180](#)
- isValid() method, [403](#)
- isVisible() method, [408](#)

J



java.awt.Dimension class, [199](#)

Javadoc application, [258](#)

JButton class hierarchy

equivalent Jython code, [22](#)

frame.add() method, [23](#)

interactive wsadmin session, [21](#)

Java code, [22](#)

wsadmin interactive session, [22](#)

JColorChooser class

components, [296](#)

constructors, [294–295](#)

sample output, [293–294](#)

JComboBox

cell editors, [175](#)

DynamicComboBox

actionPerformed() method, [89](#) BorderLayout.CENTER constant, [88](#) remove()
method, [89](#)

run() method, [87–88](#)

editable attributes, [86](#)

event.getSource() method, [85](#)

sample output, [85](#)

JDialog

class hierarchy, [264](#)

CustomDialog1.py sample output, [271](#) GraphicsConfiguration object, [267](#)

methods, [265](#)

JEditorPanes, [259](#)

JFileChooser class, [454](#)

constructors, [287](#)

dialog types, [291](#)

File Chooser dialog box, [285–286](#) FileFilter mechanism, [290–291](#)

FileSystemView instance

- descendent class, [287–288](#)
- restricted filesystem, [288–289](#) selection types, [292](#)
- JFormattedTextField, [166](#)
- FormattedTextFieldDemo.py script, [90–91](#) NumberFormat class, [91](#)
- JFrame
 - class hierarchy, [264](#)
 - classes function, [24](#)
 - classInfo() function, [25](#)
 - classInfo.py, [25](#)
 - interactive wsadmin session, [27](#)
 - java.awt.Component.doLayout() description, [26](#) Java documentation, [23](#)
 - JFrame “convenience, [29](#)
 - “layout” attributes, [27](#)
 - “layout” methods, [26](#)
 - methods and attribute names, [24–25](#)
 - wsadmin interactive session, [28](#)
- GraphicsConfiguration object, [267](#)
- JFrame add() method, [5](#)
- JFrame documentation, [19](#)
- JFrame Layout Manager, [6](#)
- JInternalFrame classes
 - advantage, [325](#)
 - events
 - eventAdapter class, [334–335](#)
 - eventLogger class, [331–332](#), [334](#)
 - iFrameEvents class, [332–333](#)
 - InternalFrameListener class, [331](#)
 - run() method, [333](#)
 - features, [329](#)
 - findNot() methods, [329](#)
 - FrameMethod application, [325–326](#)
 - iFrameEvents2.py script, [335](#)
 - vs. JFrame, [330](#)
 - makeMenu() method, [328](#)
 - run() method, [327–328](#)
 - showItems() methods, [329](#)
 - textFile(...) method, [326](#)
 - utility methods, [326–327](#)
- JLabel

- adding second label, [5](#)
- definition, [4](#)
- JFrame class, [5](#)
- window, [5](#)
- JList
 - BorderLayout frame, [112](#)
 - count() method, [109](#)
 - java.util.Vector, [105–106](#)
 - JViewport, [107](#)
 - ListSelectionListener class, [113–114](#)
 - list selection mode, [110–111](#)
 - run() method, [108–109](#)
 - ScrollPane instance, [106–107](#)
 - text input field
 - DocumentListener methods, [116](#)
 - InputMethodListener class, [114–115](#)
 - iterations, [118](#)
 - KeyListener events, [116](#)
 - keyword argument lists, [119](#)
 - run() method, [116–118](#)
 - textCheck() method, [119](#)
- JMenu entries, [125](#)
- JOptionPane methods
 - JOption show*Dialog method name variants, [274](#) showConfirmDialog() method, [279](#)
 - showInputDialog() method, [283](#)
 - showMessageDialog() method, [274](#)
 - showOptionDialog() method, [276](#)
- JPasswordField
 - ActionListener event handler method, [82–83](#) character-obfuscation property, [80–81](#)
 - event.getActionCommand() method, [82](#)
 - frame.pack() method, [80](#)
 - getPassword() method, [82](#)
 - PasswordDemo class, [79–80](#)
 - setEchoChar() method, [79](#)
 - toString() method, [84–85](#)
- Jsoup library
- javadocInfo_01.py sample output, [237](#)

- JTabbedPane, [253](#)
- simple Jsoup demonstration, [235](#)
- soupTask class, [236](#)
- steps, [234](#)
- URL, combo box to list box, [238](#)
- JSpinner class, [182](#)
- JSpinner field
 - DateFormatSymbols method, [92–93](#)
 - default spinner constructor, [94](#)
 - Spinner1 class, [92](#)
 - spinner editor, [96–97](#)
 - SpinnerModel argument, [94](#)
 - SpinnerNumberModel class
 - calendarField argument, [96](#)
 - default SpinnerDateModel, [96](#)
 - Spinner3 class, [95](#)
 - zero parameter constructor, [95](#)
 - value selection, [93–94](#)
- JSpinner renderer, [181](#)
- JSplitPane method, [251](#), [259](#)
- JTabbedPane, [253](#)
- JTable class
 - cell renderers
 - custom cell renderers, [163](#)
 - data type rendering, [163](#)
 - data type-specific cell renderers, [162](#)
 - columnSelectionAllowed and rowSelectionAllowed attributes, [158](#)
 - getTableHeader() method, [159](#)
 - individual cell selection, [158](#)
 - rowHeight Getter and Setter methods, [177](#) row selection and editing, [156](#)
 - selectionMode property, [156](#)
 - setReorderingAllowed() method, [159](#)
 - table models
 - AbstractTableModel, [160](#)
 - DefaultTableModel, [160](#)
 - getColumnClass(...) method, [160](#) isCellEditable() method, [160](#) read-only
 - TableModel class, [160](#)
 - write-only attributes, [157](#)
- JTableHeader class, [184](#)

JTree class

DefaultTreeCellEditor, [150–152](#)

DefaultTreeCellRenderer, [152](#)

makeTree() method, [149–150](#)

manipulation

buttonRow() method, [147–148](#) DynamicTree images, [146](#)

getSuffix() method, [147](#)

setLocationRelativeTo() method, [147](#)

Tree1.py script

branch node, [138](#)

cellTree() method, [139](#)

description, [137](#)

JScrollPane, [138](#)

root node, [138](#)

sample output, [138](#)

structures, [139](#)

TreeSelectionListener interface

event handler, [143–144](#)

line description, [145](#)

run() method, [144](#)

valueChanged() method, [143](#)

TreeSelectionModel class

constants, [141](#)

node selection, limitation, [142](#) run() method, [140–141](#)

selectmode attributes, [141](#)

setSelectionMode() method, [142](#) Jython, [3](#)



Keyboard events

ActionListener method, [214](#) KeyListener descendant class, [211](#) Listen3 class, [213](#)

KeyListener descendant class, [211](#) KeyListener methods, [211](#)

Keystrokes

binding (*see* Bindings)

center application, [199](#)

JTable, [197](#)

KeyBindings, [201](#)
locationRelativeTo = None, [198](#) table data, [200](#)
table properties, [200](#)

Keyword arguments, [206](#)
horizontalAlignment, [168](#)
selectionMode, [157](#)



Label
buttonPressed() method, [33–34](#) event handler, [33](#)
run() method, [33](#)
text field, [33](#)

layoutContainer() method, [60](#)
Layout Manager's show() method, [43](#) Listener methods

absolute layout technique, [222](#) adapter classes, [209](#)
ComponentEventDemo.java, [217](#) componentResized() method, [221](#) Frame1
class, [220](#)
Frame1 sample output, [220](#)
frame.getBounds() method, [221](#) input fields monitoring, [223](#)
InputVerifier, [227](#)
JButton listeners, [206](#)
keyboard events

ActionListener method, [214](#) KeyListener descendant class, [211](#) KeyListener
methods, [211](#) Listen3 class, [213](#)
Listen3.py sample output, [212](#)

Listen1.py application, [208](#)
Listen4 class, [215](#)
MouseListener methods, [208](#)
PropertyChangeListener, [224](#)

List filtering, [252](#)
ListPorts.py script file, [153](#)
ListSelectionListener class, [113–114](#) ListSelectionListener event handler, [240](#)
listServerPorts(...) method, [421](#)

Mnemonics, [131](#)

Modified cellTree() method, [430](#) MouseListener methods, [208](#)

P, Q

pane.getLayout() method, [60](#)

parseMethodHelp(...) method, [367](#)

pickATcmd(...) method, [384](#)

pickATgroup(...) method, [384](#)

PortLookupTask class, [438](#)

PortTableModel class, [439](#), [446](#)

Progress bar class

constructors, [299](#)

doInBackground(...) method, [305](#) ProgressBar0.py, [300–301](#)

propertyUpdate() method, [304](#)

sample images, [299](#)

stringPainted property, [303–304](#)

SwingWorker class

output, [301](#)

ProgressBar2.py script, [302–303](#)

ProgressMonitor class

constructors, [306](#)

interactive session, [307](#)

isCanceled() method, [310–311](#)

JDialog instances, [307–308](#)

message parameter, [311](#)

parentComponent argument, [313](#) ProgressMonitor1.py, [308–309](#)

ProgressMonitorInputStream objects, [315](#) properties, [313–314](#)

propertyUpdate() method, [309](#)

setNote(...) method, [312–313](#)

PropertyChangeListener event handler method, [224](#), [250](#)

M, N, O

MenuBar(...) method, [442](#)

Menus

accelerator, [132](#)

actionPerformed() method, [126](#) addActionListen() method, [126](#) adding
MenuBar, [122](#)
check boxes, [129](#)
class hierarchy, [121](#)
contrasting menu entries, [125](#)
empty JMenuBar, [123](#)
foreground and background colors, [124](#) MenuBar methods, [122](#)
mnemonics, [131](#)
pop-up menu, [134](#)
radio buttons, [127](#)
Merriam Webster page, [233](#)
MessageType constants, [275](#)



readFileSAX routine, [450](#)
Reflection, [4](#)
removeChoosableFileFilter() method, [291](#) reportConfiguredPorts(...) method,
[422](#) reportTableModel, [408](#)
Rigid area creation method, [57](#)
Row filtering, [406](#)
rowFinder class, [403](#)
run() method, [32](#), [59](#), [404](#)



SaveTask and DiscardTask classes, [445](#) SAXhandler methods, [450](#)
ScreenLoc class, [268](#)
ScreenPos class, [268](#)
Scripting report method, [387](#)
Security configuration report

administration console, [385](#)
AdminTask method, [416](#)
code changes, [409](#)
column widths

output, [399](#)
processReport routine, [397](#) setColumnWidths() method, [397](#) WebSphere

Application Server, [397](#)

frame resize listener, [400](#)

modifications, [390](#)

quick and dirty attempt, [389](#)

rowFinder class, [403](#)

scripting report method, [387](#)

section visibility

clicker(...) method, [413](#)

Find(...) method, [414](#)

modified clicker(...) method, [415](#) reportTableModel, [412](#)

scopeName, [411](#)

sectionFilter class, [412](#)

testing, [415](#)

table model and cell renderer

HTML coloring, [394](#)

multiple selection issue, [392](#) revised cell renderer, [395](#)

script, [392](#)

table row filtering, [406](#)

upDownAction class, [404](#)

Selectable input components

check boxes, [100–101](#)

radio buttons, [101–102](#)

toggle buttons

application window, [99](#)

ButtonGroupDemo, [103](#)

getItem() method, [100](#)

JToggleButton constructor, [99](#)

setBounds() method, [36](#)

setColumnWidths() method, [397](#), [441](#) setDividerLocation(...) methods, [251](#)

setDividerSize() method, [47](#)

setFileSelectionMode() method, [292](#) setHighlighter(...) method, [362](#)

setProgress(...) method, [314](#)

setRightComponent(...) method, [259](#) setSelectionMode() method, [111](#)

setText(...) method, [251](#)
setValidating(...) method, [452](#)
setValueAt(...) method, [172](#), [174](#)
setVisible() method, [408](#)
showConfirmDialog() method, [279](#)
showDialog() method, [291](#)
showInputDialog() method, [283](#)
showMessageDialog() method, [274](#)
showOpenDialog() method, [291](#)
showOptionDialog() method, [276](#)
showSaveDialog() method, [291](#)
SimpleDialog class, [270](#)
SplitPane PropertyChangeListener output, [250](#) Split panes

application, [44](#)
component size
attributes, [46](#)
and divider bar, [49](#)

horizontal separation, [44](#)
limitation, [44](#)
nested split panes, [47](#)
OneTouchExpandable divider, [48](#)
oneTouchExpandable keyword argument, [48](#) setDividerSize() method, [47](#)
vertical splits, [44](#)

startElement(...) method, [451](#), [456](#)

Swing threads
equivalent approach, [13](#)
Runnable class, [12](#)
Swing component-creation operations, [13](#) SwingUtilities/EventQueue class, [12](#)
template script, [12–13](#)

SwingWorker setProgress(...) method, [302](#)



TabbedPane, [49](#)
Table1.py output, [154](#)

Text input fields

admin console inactivity timeout, [65](#) AvailableFonts class, [74](#)

consoleTimeout1 class, [67](#)

JTextArea

caretUpdate(), [75](#)

listeners, [75](#)

setEditable() method, [74](#) SimpleEditor class, [76](#)

JTextField

ActionListener, [66](#)

TextAlignment class, [71–72](#)

SwingWorker class

concurrency, [68](#)

output, [69](#)

subclass, [69](#)

threads, [68](#)

Text input fields (*cont.*)

update() method, [68](#), [70](#)

WSAStask class, [69–70](#)

wsadmin application, [66](#)

Top-level containers, [1](#)

treeNodesChanged() method, [151](#) TreeSelectionListener class, [430](#), [435](#), [437](#)

❖❖❖U

update() method, [345](#)

❖❖❖❖❖❖❖❖❖

valueChanged(...) event handler method, [375](#) valueChanged(...) method, [374](#)

❖❖❖❖❖❖❖❖❖W, X, Y, Z

WAShome(...) routine, [435](#)

WASprofileName(...) routine, [435](#)

WASvarLookup(...) routine, [435](#)

WASversion(...) routine, [435](#)

WebSphere Application Server, [397](#)

WebSphere Port (WASports) application

- AdminConfig methods, [423](#)
- administration console, [419](#)
- AdminTask.listServerPorts() method, [420](#)
- AdminTask.reportConfiguredPorts(...) method, [422](#) cell and node tree items

- cellTSL class, [435](#)
- sample output, [432](#)
- utility routines, [432](#)

- configurations, [461](#)
- creation, [425](#)
- empty internal frame, [425](#)
- export functionality

- API (*see* Application Programming Interface (API))
- DTD, [452](#)
- ExportTask class, [453](#)
- xml.dom.minidom, [448](#)
- import functionality, [457](#)
- JSplitPane
 - cell hierarchy tree, [427](#)
 - internal frame, [426](#)
 - menu items, [442](#)
 - report generation, [461](#)
 - save and discard, [444](#)
 - serve port number
- PortLookupTask class, [439](#)
- PortTableModel class, [439](#)
- Sample output, [436](#)
- TreeSelectionListener class, [437](#)
- utility routine, [437](#)
- split pane, [428](#)
- table column width, [440](#)
- table sorting, [461](#)
- text highlighting, [461](#)
- WebSphere Port(WASports) application WindowAdapter class, [447](#)
- WSAShelp application
 - adding menu, [376](#)

- adding split panes, [358](#)
- AdminTask.help('-commandGroups') adding menu, [383](#)
- AdminTask.help(...) method, [379](#) AdminTask help text, [383](#)
- ATcommandTask class, [379](#), [381](#) ATgroupsTask class, [380–381](#)
- clusterConfig step, [384](#)
- implementation, [382](#)
- pickATcmd(...) method, [384](#)
- pickATgroup(...) method, [384](#) showCmdGroups(...) method, [381](#) split pane, [383](#)
- SwingWorker descendent class, [380](#) text pane selectable, [383](#)
- AdminTask.help('-commands')
- AdminTask menu item, [378](#)
- doInBackground(...) method, [377](#) output, [377](#)
- “Show” menu item, [378](#)
- SwingWorker class, [377](#)
- cellSelector listener class, [375](#)
- displaying methods
- camelWords(...) method, [372](#) DefaultHighlighter class, [367](#)
- getFontMetrics(...) method, [371](#)
- getTableCellRendererComponent(...) method, [372](#)
- highlight table text, [367](#)
- installInteractive method, [375](#)
- interactive wsadmin session, [371](#)
- JTable, [371](#)
- JTextPane component, [367](#)
- methodTableModel class, [368–369](#)
- MethodTable scripts, [368](#)
- methRenderer class, [371–372](#)
- output, [370](#)
- parseMethodHelp(...) method, [366](#)
- proof of concept (PoC), [366](#), [368](#)
- run(...) method, [366](#)
- setColumnWidths(...) method, [369](#)
- setHiText(...) method, [372](#)
- setWidths(...) method, [372–373](#) Help.help() method, [355](#)
- ListSelectionListener class, [373](#)
- output, [356](#)
- scripting object, [374–375](#)
- tabbed pane, [357](#)
- text highlighting

ActionListener event handler, [361](#) advantage, [361](#)
ChangeListener method, [363](#) DefaultHighlighter class, [362](#)
DefaultHighlightPainter class, [361](#) find(...) method, [362](#)
Help.wsadmin() method, [360](#)
Highlight class, [360](#)
search(...) method, [361–362](#)
setHighlighter(...) method, [362](#) tabbed highlighting complications, [363](#)
valueChanged(...) event handler
method, [375](#)
valueChanged(...) method, [374](#)

Swing for Jython

**Jython UI and Scripts Development using Java Swing and
WebSphere Application Server**

Robert A. Gibson

**Swing for Jython: Graphical Jython UI and Scripts Development using
Java Swing and WebSphere Application Server**

Copyright © 2015 by Robert Gibson

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0818-2

ISBN-13 (electronic): 978-1-4842-0817-5

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an

expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Steve Anglin

Development Editor: Tracy Brown Hamilton

Technical Reviewers: Rohan Walia, Dhrubojyoti Kayal, Manuel Jordan Elera,
and Frank Wierzbicki Editorial Board: Steve Anglin, Louise Corrigan, Jonathan
Gennick, Robert Hutchinson, Michelle Lowman,

James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick,
Ben Renow-Clarke, Gwenan Spearing, Steve Weiss

Coordinating Editor: Mark Powers

Copy Editor: Kezia Endsley

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media
New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-
SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit
www.springeronline.com. Apress Media, LLC is a California LLC and the sole
member (owner) is Springer Science + Business Media Finance Inc (SSBM
Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit
www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic,
corporate, or promotional use. eBook versions and licenses are also available for
most titles. For more information, reference our Special Bulk Sales—eBook
Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this

text is available to readers at www.apress.com/9781484208182. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

I thank God for his countless gifts and blessings and dedicate this work to my bride Linda, and our children. Thank you for loving me, and for putting up with me and my fondness of puns. I'm sorry for all the time that this has required, but I have thought of you all throughout its development. I also thank everyone who helped make this book a reality. I could not have done this without your assistance, nor would it have been anywhere as good as you have helped make it.

Contents

About the Author

xix About the Technical Reviewers

? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?

xxi Introduction

Introduction

xxiii

■ Chapter 1: Components and

Containers

Top-Level

Containers 

Getting Help from Jython

How and Why Are You Able to Do This?

What's Next?...Starting Simple

Adding a Second

Label 

Summary 

■ Chapter 2: Interactive Sessions vs ?

Scripts 

Running Your First Script from a

File 

Depending “Too Much” on Limited Information

Swing

Threads

Summary

■ Chapter 3: Building a Simple Global Security

Application

Adding Text to the Application Using a

JLabel

No “Pane,” No

Gain

When You Live in a Glass House, Everything Is a

Pane

16

The Layered Look Can Also Be a

Pane

17

vii The Optional MenuBar

17

The Content Pane Will Contain Most of the Visible Items

17

Summary

■ Chapter 4: Button Up! Using Buttons and

Labels

JButton Class Hierarchy

The Layout of the

Land

Buttons! Labels! Action!

Updating the Application

Summary

■ Chapter 5: Picking a Layout Manager Can Be a

Pane

The Absolute Layout Manager Does Not Corrupt

Absolutely

Going with the Flow: The FlowLayout

Manager

South of the Border: The BorderLayout Manager

What's in the Cards? Using the CardLayout

Manager

Splitting Up Is Easy to Do: Using Split

Panes

Vertical Splits: Not as Painful as They Sound

44

Limited Resources: Setting Size

Attributes

46

Nested Split Panes

47

Divider and Conquer

47

Rules for Using Split Panes

49

Can I Run a Tab? Using a

TabbedPane

Are You Boxed In? Using the BoxLayout

Manager

The Box Class

53

Building a

Box

54

Invisible Box Components

55

Boxes and Resizable

Components

57

viii

Gridlock, Anyone? Using the GridLayout Manager

58

Shaking Things Up: The GridBagLayout

Manager

Looking at Other Layout Managers

Summary

■Chapter 6: Using Text Input

Fields

What Does It Take to Get Data Into an Application?

JTextField: Getting Data Into the

Application

Your First, Almost Real,

Application

Help Me SwingWorker, You're My Only Hope!

Back to the JTextField

Size Matters: Looking at Text Font Attributes

72

The Elephant (Font) in the Room

Using JTextArea for Input

Summary

■Chapter 7: Other Input

Components

Password Fields

Is There an Echo in Here? Using the Character-Obfuscation

Property 80

The getPassword()
Method 82

The event getActionCommand() Method
82

The JPasswordField Event Handler
82

Converting jarray Values to Strings
84

Choosing from a List
Editing a
ComboBox 86

Using the DynamicComboBox

Formatted Text Fields

Using a JSpinner Text Field

Some DateFormatSymbols Methods
92

The JSpinner Class
93

The SpinnerModel Class
94 The JSpinner Editor

96
Summary

■Chapter 8: Selectable Input
Components

Toggle

Buttons

Check

Boxes

Radio Buttons

Toggle Buttons in a Button

Group

Summary

■Chapter 9: Providing Choices, Making

Lists

Making a List and Checking It

Twice

Optional Scroll

Bars

The ScrollPane

Viewport

107 Manipulating the

List

Counting List

Items

108 Limiting the Selectable Items

110 Reacting to List-Selection

Events

111 Reacting to User (Text) Input

114

Summary

■Chapter 10: Menus and

MenuItems

The JMenu Class

Hierarchy

Reacting to Menu-Related Events

Using Radio Buttons on a

Menu

Using Check Boxes on a Menu

Menus 

Menus 

[illegible]

■Chapter 11: Using JTree to Show the Forest: Hierarchical Relationships of Components

JTree Attributes and Methods

Class

140

JTree
Manipulation

The DefaultTreeModel Class

A horizontal row of 20 identical black diamond-shaped icons. Each icon contains a white question mark.

The TreeModelListener Interface

150

Summary

■Chapter 12: Motion to Take from the Table: Building
Tables

Tables Can Be Really

Easy 

Defaults Can Be Harmful to your Mental

Health 

Picky, Picky, Picky? Selecting Parts of a

Table 155

Row, Row, Row Your Table? Working with Rows 156

Selecting Columns 158

Selecting Individual Cells 158

I Am the Very Model of a Modern Major General: Table Models

Types of Table Models 159

Cell Renderers

Custom Cell Renderers 163

Using Cell Editors

Boolean Cell Editors 172

Numeric Cell Editors 173

Custom Numeric Cell Editors 174

JComboBox Cell Editors 175

Warning: Ugliness Ahead 178

Column Manipulation

Column Widths	183
Column Heading Width	184
Determining Column Width	185
Column Adjustments	188
Summary	
■Chapter 13: Keystrokes, Actions, and Bindings, Oh My!	
Getting in a Bind: Looking at Bindings	
What Is Meant by Binding?	
193 InputMaps and ActionMaps	
193 JTable Keystrokes	
Putting It All Together	
locationRelativeTo =	
None	
198 Centering the Application	
199 Defining the Table	
Properties	
200 Computing the Table	
Data	
200 The Fruits of Your Labor	
201 Binding Reuse	
Where to Begin: Finding the Appropriate Action Class	

203 What Do You Need to Worry About? Boundary

Conditions

203

Summary

■Chapter 14: It's the Event of the Year: Events in Swing

Applications

If an Event Occurs and No One Hears It

Using Listener

Methods

Put Your Listener Where Your Component Is

Adapt or Die: Using Adapter Classes

Listening for Keyboard

Events

Most Objects Never Really

Listen

Looking for a Listener in a Haystack

Using a ComponentAdapter to Monitor Changes

Monitoring the Input Fields

xii

Using a PropertyChangeListener

224

Using an

InputVerifier

Summary

■Chapter 15: Nuts to Soup: Using Jsoup to Enhance

Applications

Using Existing Classes: Creating an HTML Retrieval Application from

Scratch

Wouldn't It Be Nice: Using Java

Libraries

231

Working with the Jsoup Library

Jsoup Call May Appear to Hang

From a Combo Box to a List Box

Adding a TextArea to Show the HTML

Rendering

HTML

Modifying the HTML Text

Identifying the

Sections

Fixing the Great Divide

Filtering the List

Using Jsoup to Pick Up the Tab: Adding a JTabbedPane

Adding Tabbed Editor Panes to the Javadoc

Application

What Improvements/Enhancements Remain?

Summary

■Chapter 16: Conversing with a User with Dialog Boxes

What Are Dialog Boxes?

What's a JDialog?

What's the GraphicsConfiguration Component Do?

Using a GraphicsConfiguration

Object

Is a GraphicsConfiguration Object Really Necessary?

What About an Owner?

Annoying 

Using JOptionPane Methods

The JOptionPane.showMessageDialog()

Method 

The JOptionPane.showOptionDialog()

Method

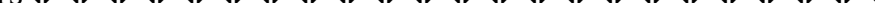
The JOptionPane.showConfirmDialog() Method

The JOptionPane.showInputDialog()

Method [illegible]

Boxes 

A horizontal sequence of 20 identical black diamond shapes, each containing a white question mark.

Constructors 

287

Using a Filesystem View

287

File

Filtering

290

Chooser Dialog

Types

291

Selection

Types 
[292](#)

The JColorChooser

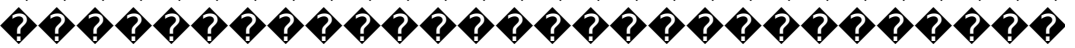
Class 

The javax.swing.colorchooser

Package 
[294](#)

What Else Do You Need to Know About These “Special” Dialog Boxes?



Summary 

■ Chapter 18: Monitoring and Indicating Progress



Changing the

Cursor 

Showing a Progress

Bar 

SwingWorker Progress



[301](#)

Showing Progress

Details 

[303](#)

Specifying a Progress Bar Range



[304](#)

Indeterminate ProgressBar

Range 

[305](#)

ProgressMonitor

Objects 

ProgressMonitor Cancellation



[309](#)

The ProgressMonitor Message



[311](#)

The ProgressMonitor

Note 

312

The ProgressMonitor

parentComponent

313 Other ProgressMonitor Properties

313 ProgressMonitorInputStream Objects

Summary

■Chapter 19: Internal

Frames

Looking at Inner Frames

Layers

Position Within the Layer

321 The JDesktopPane Class

JFrame or JInternalFrame?

The JInternalFrame

Class

JFrame and JInternalFrame

Methods

325 More JFrame and JInternalFrame Differences

330 JInternalFrame

Events

331 More JInternalFrame Topics

335 Building an Application from

Scratch

Simple Non-GUI Jython Version of the consoleTimeout Script

336 First GUI Jython Version of the consoleTimeout

Script

337 Adding Menu

Items

341 Changing from JFrame to

JInternalFrame

342

Summary

■ Chapter 20: Building a Graphical Help

Application

Showing the Help

Text

Using a Tabbed Pane

Adding Split Panes

Text

Highlighting

Adding Text Highlighting to WSAshelp

Application

362 Tabbed Highlighting Complications

363

xv

Displaying Methods in a

Table

365

Highlighting Text Within the

Table

367

Using Tables in the Help

Application

368

Fixing the Table

Appearance

371

Selecting Table Cells

Adding a Menu

AdminTask help('-commands'

)

How Do We Find and Identify Existing Commands?

377

[illegible]

AdminTask❖help('-commandGroups')

A horizontal row of 28 identical black diamond shapes, each containing a white question mark.

A horizontal row of 28 diamond-shaped icons, each containing a question mark.

Step It Up: Displaying the “Steps” Help Text

Should You Add a Menu?

Should You Add Another Split Pane?

Can You Make Parts of the Text Pane Selectable?

And Now for Something Completely Different???

384

[illegible]

■Chapter 21: A Security Configuration Report

Application 

Generating the Administration Console Report

The Scripting Report Method

First

Attempt 

Second Attempt, Ignoring the Last

Delimiter 

Adding a Table Model and Cell

Renderer

Using Color Instead of a Bold Font

394

Adjusting Column Widths

Column Widths and Row

Heights

397

Adding a Frame Resize

Listener

Fixing the Row Selection Colors

402

Which Rows Are Visible?

403

Table Alignment in the

Viewport

Table Row

Filtering

Finding Text

Section Visibility

Does It Work?

414

Progress Indicator

Summary

■Chapter 22: WASports: A WebSphere Port

Application

Using the Administration Console

The AdminTask

Method

The AdminTask

452	Initial WASports
452	DTD
453	The ExportTask
453	Class
	Step 11: Implementing the Import Functionality
	The ImportTask Class
458	
	What's Left?
	Text
	Highlighting
461	
	Table
	Sorting
461	
	Comparing Configurations
461	
	Report
	Generation
461	
	Summary
	Index
xviii	

About the Author

Robert A. (Bob) Gibson is an Advisory Software Engineer with decades of experience in numerous software-related roles at IBM, including Architect, Developer, Tester, Instructor, and Technical Support. While providing technical support for the IBM's WebSphere Application Server product, he was the primary author for "WebSphere Application Server Administration Using

Jython” which was published by IBM Press. He is currently a member of the IBM technical support team responsible for the IBM MQ product on distributed platforms. He holds both a Bachelor of Science degree in Engineering Science and a Master of Science degree in Computer Science from the University of Virginia.

About the Technical Reviewers



Manuel Jordan Elera is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations.

Manuel won the 2010 Springy Award – Community Champion and Spring

Champion 2013. In his little free time, he reads the Bible and composes music on his guitar. Manuel is known as dr_pompeii. He has tech reviewed numerous books for Apress, including *Pro Spring, 4th Edition* (2014), *Practical Spring LDAP* (2013), *Pro JPA 2, Second Edition* (2013), and *Pro Spring Security* (2013).

Read his 13 detailed tutorials about many Spring technologies, contact him through his blog at <http://www.manueljordanelera.blogspot.com>, and follow him on his Twitter account, @dr_pompeii

Dhrubojyoti Kayal is a hands-on Java developer and architect. He is an open source evangelist. He has been helping enterprises solve integration challenges and build complex large applications leveraging Java technologies for the past 14 years. His current area of focus is data migration and real-time analytics with Java. Dhrubojyoti is also the author of *Pro Java EE Spring Pattern* (2008) from Apress.

Rohan Walia is a Senior Software Consultant with extensive experience in client/ server, web-based, and enterprise application development. He is an Oracle Certified ADF Implementation Specialist and a Sun Certified Java Programmer. Rohan is responsible for designing and developing end-to-end applications consisting of various cutting-edge frameworks and utilities. His areas of expertise are Oracle ADF, Oracle WebCenter, Fusion, Spring, Hibernate, and Java/J2EE. When not working, Rohan loves to play tennis, hike, and travel. Rohan would like to thank his wife, Deepika Walia, for using all her experience and expertise when reviewing this book.

■ about the teChniCal RevieweRs



xxii

Frank Wierzbicki is the head of the Jython project and a member of the Python

Software Foundation. He has over 15 years of experience as a software developer, primarily working in Python and Java. He has been programming since the Commodore 64 was the king of home computers (look it up kids!) and can't imagine why anyone would do anything else for a living. Frank's most enduring hobby is picking up new programming languages, but he has yet to find one that is more fun to work with than Python.



Quick answers to common problems

Python Data Visualization Cookbook

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

Igor Milovanović

[PACKT] open source*
PUBLISHING community experience distilled

Python Data Visualization Cookbook

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

Igor Milovanović BIRMINGHAM - MUMBAI

Python Data Visualization Cookbook

Copyright © 2013 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: November 2013
Production Reference: 1191113

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

ISBN 978-1-78216-336-7
www.packtpub.com
Cover Image by Gorkee Bhardwaj (afterglowpictures@gmail.com)

Credits

Author

Igor Milovanović Project Coordinator Rahul Dixit

Reviewers

Tarek Amr

Simeone Franklin Jayesh K. Gupta

Kostiantyn Kucher Kenneth Emeka Odoh

Proofreaders

Amy Johnson Lindsey Thomas

Indexer

Mariammal Chettiyar

Acquisition Editor James Jones

Graphics

Abhinash Sahu

Lead Technical Editor Ankita Shashi

Production Coordinator Shantanu Zagade

Technical Editors Pratik More

Amit Ramadas Ritika Singh

Copy Editors

Brandt D'Mello

Janbal Dharmaraj Deepa Nambiar Kirti Pai

Laxmi Subramanian

Cover Work

Shantanu Zagade

About the Author

Igor Milovanović is an experienced developer with a strong background in Linux system and software engineering. He has skills in building scalable data-driven distributed software-rich systems.

He is an Evangelist for high-quality systems design who holds strong interests in software architecture and development methodologies. He is always persistent on advocating methodologies that promote high-quality software, such as test-

driven development, one-step builds, and continuous integration.

He also possesses solid knowledge of product development. Having field experience and official training, he is capable of transferring knowledge and communication flow from business to developers and vice versa.

I am most grateful to my fiancé for letting me spend endless hours on the work instead with her and for being an avid listener to my endless book monologues. I want to also thank my brother for always being my strongest supporter. I am thankful to my parents for letting me develop myself in various ways and become the person I am today.

I could not write this book without enormous energy from open source community that developed Python, matplotlib, and all libraries that we have used in this book. I owe the most to the people behind all these projects. Thank you.

About the Reviewers

Tarek Amr achieved his postgraduate degree in Data Mining and Information Retrieval from the University of East Anglia. He has about 10 years' experience in Software Development. He has been volunteering in Global Voices Online (GVO) since 2007, and currently he is the local ambassador of the Open Knowledge Foundation (OKFN) in Egypt. Words such as Open Data, Government 2.0, Data Visualisation, Data Journalism, Machine Learning, and Natural Language Processing are like music to his ears.

Tarek's Twitter handle is @gr33ndata and his homepage is <http://tarekamr.appspot.com/>.

Jayesh K. Gupta is the Lead Developer of Matlab Toolbox for Biclustering Analysis (MTBA). He is currently an undergraduate student and researcher at IIT Kanpur. His interests lie in the field of pattern recognition. His interests also lie in basic sciences, recognizing them as the means of analyzing patterns in nature. Coming to IIT, he realized how this analysis is being augmented by Machine Learning algorithms with various diverse applications. He believes that augmenting human thought with machine intelligence is one of the best ways to advance human knowledge. He is a long time technophile and a free-software Evangelist. He usually goes by the handle, rejuvyesh online. He is also an avid reader and his books can be checked out at Goodreads. Checkout his projects at

Bitbucket and GitHub. For all links visit [http:// home.iitk.ac.in/~jayeshkg/](http://home.iitk.ac.in/~jayeshkg/). He can be contacted at a2z.jayesh@gmail.com.

Kostiantyn Kucher was born in Odessa, Ukraine. He received his Master's degree in Computer Science from Odessa National Polytechnic University in 2012. He used Python as well as Matplotlib and PIL for Machine Learning and Image Recognition purposes.

Currently, Kostiantyn is a PhD student in Computer Science specializing in Information Visualization. He conducts his research under the supervision of *Prof. Dr. Andreas Kerren* with the ISOVIS group at the Computer Science Department of Linnaeus University (Växjö, Sweden).

Kenneth Emeka Odoh performs research on state of the art Data Visualization techniques. His research interest includes exploratory search where the users are guided to their search results using visual clues.

Kenneth is proficient in Python programming. He has presented a Python conference talk at Pycon, Finland in 2012 where he spoke about Data Visualization in Django to a packed audience.

He currently works as a Graduate Researcher at the University of Regina, Canada. He is a polyglot with experience in developing applications in C, C++, Python, and Java programming languages.

When Kenneth is not writing source codes, you can find him singing at the Campion College chant choir.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.

TM

<http://PacktLib.PacktPub.com> Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

f Fully searchable across every book published by Packt f Copy and paste, print and bookmark content f On demand and accessible via 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 nine entirely free books. Simply use your login credentials for immediate access.

Preface 1

Chapter 1: Preparing Your Working Environment 5

Introduction 5

Installing matplotlib, NumPy, and SciPy 6

Installing virtualenv and virtualenvwrapper 8

Installing matplotlib on Mac OS X 10

Installing matplotlib on Windows 11

Installing Python Imaging Library (PIL) for image processing 12

Installing a requests module 14

Customizing matplotlib's parameters in code 14

Customizing matplotlib's parameters per project 16

Chapter 2: Knowing Your Data 19

Introduction 19

Importing data from CSV 20

Importing data from Microsoft Excel files 22

Importing data from fixed-width datafiles 25

Importing data from tab-delimited files 27

Importing data from a JSON resource 28

Exporting data to JSON, CSV, and Excel 31

Importing data from a database 36

- Cleaning up data from outliers 40
- Reading files in chunks 46
- Reading streaming data sources 48
- Importing image data into NumPy arrays 50
- Generating controlled random datasets 56
- Smoothing the noise in real-world data 64

Chapter 3: Drawing Your First Plots and Customizing Them 71

- Introduction 72
- Defining plot types – bar, line, and stacked charts 72
- Drawing a simple sine and cosine plot 78
- Defining axis lengths and limits 81
- Defining plot line styles, properties, and format strings 84
- Setting ticks, labels, and grids 89
- Adding a legend and annotations 92
- Moving spines to the center 95
- Making histograms 96
- Making bar charts with error bars 99
- Making pie charts count 101
- Plotting with filled areas 103
- Drawing scatter plots with colored markers 105

Chapter 4: More Plots and Customizations 109

- Introduction 109
- Setting the transparency and size of axis labels 110
- Adding a shadow to the chart line 113
- Adding a data table to the figure 116
- Using subplots 118
- Customizing grids 121
- Creating contour plots 125
- Filling an under-plot area 128
- Drawing polar plots 131
- Visualizing the filesystem tree using a polar bar 134

Chapter 5: Making 3D Visualizations 139

- Introduction 139
- Creating 3D bars 139
- Creating 3D histograms 143
- Animating in matplotlib 146
- Animating with OpenGL 150

Chapter 6: Plotting Charts with Images and Maps	157
Introduction	157
Processing images with PIL	158
Plotting with images	164
Displaying an image with other plots in the figure	168
Plotting data on a map using Basemap	172
Plotting data on a map using Google Map API	177
Generating CAPTCHA images	183
Chapter 7: Using Right Plots to Understand Data	189
Introduction	189
Understanding logarithmic plots	190
Understanding spectrograms	193
Creating a stem plot	198
Drawing streamlines of vector flow	201
Using colormaps	205
Using scatter plots and histograms	210
Plotting the cross-correlation between two variables	217
Importance of autocorrelation	220
Chapter 8: More on matplotlib Gems	225
Introduction	225
Drawing barbs	225
Making a box and a whisker plot	229
Making Gantt charts	232
Making errorbars	237
Making use of text and font properties	240
Rendering text with LaTeX	246
Understanding the difference between pyplot and OO API	250
Index	257

The best data is the data that we can see and understand. As developers, we want to create and build the most comprehensive and understandable visualizations. It is not always simple; we need to find the data, read it, clean it, massage it, and then use the right tool to visualize it. This book explains the process of how to read, clean, and visualize the data into information with straight and simple (and not so simple) recipes.

How to read local data, remote data, CSV, JSON, and data from relational databases are all explained in this book.

Some simple plots can be plotted with a simple one-liner in Python using matplotlib, but doing more advanced charting requires knowledge of more than just Python. We need to understand the information theory and human perception aesthetics to produce the most appealing visualizations.

This book will explain some practices behind plotting with matplotlib in Python, statistics used, and usage examples for different charting features we should use in an optimal way. This book is written and the code is developed on Ubuntu 12.03 using Python 2.7, IPython 0.13.2, virtualenv 1.9.1, matplotlib 1.2.1, NumPy 1.7.1, and SciPy 0.11.0.

What this book covers

Chapter 1, Preparing Your Working Environment, covers a set of installation recipes and advices on how to install the required Python packages and libraries on your platform. *Chapter 2, Knowing Your Data*, introduces you to common data formats and how to read and write them, be it CSV, JSON, XSL, or relational databases.

Chapter 3, Drawing Your First Plots and Customizing Them, starts with drawing simple plots and covers some of the customization.

Chapter 4, More Plots and Customizations, follows up from previous chapter and covers more advanced charts and grid customization.

Chapter 5, Making 3D Visualizations, covers three-dimensional data visualizations such as 3D bars, 3D histograms, and also matplotlib animations.

Chapter 6, Plotting Charts with Images and Maps, covers image processing, projecting data onto maps, and creating CAPTCHA test images.

Chapter 7, Using Right Plots to Understand Data, covers explanations and recipes on some more advanced plotting techniques such as spectrograms and correlations. *Chapter 8, More on matplotlib Gems*, covers a set of charts such as Gantt charts, box plots, and whisker plots, and also explains how to use LaTeX for rendering text in matplotlib.

What you need for this book

For this book, you will need Python 2.7.3 or a later version installed on your operating system. This book was written using Ubuntu 12.03's Python default version (2.7.3).

Other software packages used in this book are IPython, which is an interactive Python environment that is very powerful, and flexible. This can be installed using package managers for Linux-based OSes or prepared installers for Windows and Mac OSes.

If you are new to Python installation and software installation in general, it is very much recommended to use prepackaged scientific Python distributions such as Anaconda, Enthought Python Distribution, or Python(X,Y).

Other required software mainly comprises of Python packages that are all installed using the Python installation manager, pip, which itself is installed using Python's easy_install setup tool.

Who this book is for

Python Data Visualization Cookbook is for developers who already know about Python programming in general. If you have heard about data visualization but don't know where to start, this book will guide you from the start and help you understand data, data formats, data visualization, and how to use Python to visualize data.

You will need to know some general programming concepts, and any kind of programming experience will be helpful. However, the code in this book is explained almost line by line. You don't need math for this book; every concept that is introduced is thoroughly explained in plain English, and references are available for further interest in the topic.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning. Code words in text are shown as follows: "We packed our little demo in class DemoPIL, so that we can extend it easily, while sharing the common code around the demo function, run_fixed_filters_demo."

A block of code is set as follows: `def _load_image(self, imfile): self.im = mpltimage.imread(imfile)` When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# tidy up tick labels size
```

```
all_axes = plt.gcf().axes
for ax in all_axes:

    for ticklabel in ax.get_xticklabels() + ax.get_yticklabels():
        ticklabel.set_fontsize(10)
```

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

```
$ sudo python setup.py install
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "We then set up a label for the stem plot and the position of baseline, which defaults to 0."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

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

1

Preparing Your Working Environment

In this chapter, we will cover the following recipes:

- f Installing matplotlib, NumPy, and SciPy
- f Installing virtualenv and virtualenvwrapper
- f Installing matplotlib on Mac OS X
- f Installing matplotlib on Windows
- f Installing Python Imaging Library (PIL) for image processing
- f Installing a requests module
- f Customizing matplotlib's parameters in code
- f Customizing matplotlib's parameters per project

Introduction

This chapter introduces the reader to the essential tooling and installation and configuration of them. This is a necessary work and common base for the rest of the book. If you have never used Python for data and image processing and visualization, it is advised not to skip this chapter. Even if you do skip it, you can always return to this chapter in case you need to install some supporting tool or verify what version you need to support the current solution.

Installing matplotlib, NumPy, and SciPy

This chapter describes several ways of installing matplotlib and required dependencies under Linux.

Getting ready

We assume that you already have Linux (preferably Debian/Ubuntu or RedHat/SciLinux) installed and Python installed on it. Usually, Python is already installed on the mentioned Linux distributions and, if not, it is easily installable through standard means. We assume that Python 2.7+ Version is installed on your workstation.

Almost all code should work with Python 3.3+ Versions, but because most operating systems still deliver Python 2.7 (some even Python 2.6) we decided to write the Python 2.7 Version code. The differences are small, mainly in version of packages and some code (xrange should be substituted with range in Python 3.3+).

We also assume that you know how to use your OS package manager in order to install software packages and know how to use a terminal.

Build requirements must be satisfied before matplotlib can be built.

matplotlib requires NumPy, libpng, and freetype as build dependencies. In order to be able to build matplotlib from source, we must have installed NumPy.

Here's how to do it: Install NumPy (at least 1.4+, or 1.5+ if you want to use it with Python 3) from <http://www.numpy.org/>.

NumPy will provide us with data structures and mathematical functions for using it with large datasets. Python's default data structures such as tuples, lists, or dictionaries are great for insertions, deletions, and concatenation. NumPy's data structures support "vectorized" operations and are very efficient for use and for executions. They are implemented with Big Data in mind and rely on C implementations that allow efficient execution time.

SciPy, building on top of NumPy, is the de facto standard's scientific and numeric toolkit for Python comprising great selection of special functions and algorithms, most of them actually implemented in C and Fortran, coming from the well-known Netlib repository (see <http://www.netlib.org>). Perform the following steps for installing NumPy:

1. Install Python-NumPy package:

\$ sudo apt-get install python-numpy 2. Check the installed version:

\$ python -c 'import numpy; print numpy.__version__'

3. Install the required libraries: ♦ libpng 1.2: PNG files support (requires zlib)

♦ freetype 1.4+: True type font support

\$ sudo apt-get install build-dep python-matplotlib If you are using RedHat or variation of this distribution (Fedora, SciLinux, or CentOS) you can use yum to perform same installation:

\$ su -c 'yum-builddep python-matplotlib'

How to do it...

There are many ways one can install matplotlib and its dependencies: from source, from precompiled binaries, from OS package manager, and with prepackaged python distributions with built-in matplotlib.

Most probably the easiest way is to use your distribution's package manager. For Ubuntu that should be:

in your terminal, type:

\$ sudo apt-get install python-numpy python-matplotlib python-scipy

If you want to be on the bleeding edge, the best option is to install from source. This path comprises a few steps: Get the source, build requirements, and configure, compile, and install.

Download the latest source from code host www.github.com by following these steps:

\$ cd ~/Downloads/

\$ wget

<https://github.com/downloads/matplotlib/matplotlib/matplotlib1.2.0.tar.gz>

\$ tar xzf matplotlib-1.2.0.tar.gz

\$ cd matplotlib-1.2.0

\$ python setup.py build

\$ sudo python setup.py install

Downloading the example code

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

How it works...

We use standard Python Distribution Utilities, known as Distutils, to install matplotlib from source code. This procedure requires us to previously install dependencies, as we already explained in the *Getting ready* section of this recipe. The dependencies are installed using the standard Linux packaging tools.

There's more...

There are more optional packages that you might want to install depending on what your data visualization projects are about.

No matter what project you are working on, we recommend installing IPython—an Interactive Python shell that supports PyLab mode where you already have matplotlib and related packages, such as NumPy and SciPy, imported and ready to play with! Please refer to IPython's official site on how to install it and use it—it is, though, very straightforward.

Installing virtualenv and virtualenvwrapper

If you are working on many projects simultaneously, or even just switching between them frequently, you'll find that having everything installed system-wide is not the best option and can bring problems in future on different systems (production) where you want to run your software. This is not a good time to find out that you are missing a certain package or have versioning conflicts between packages that are already installed on production system; hence, virtualenv.

virtualenv is an open source project started by *Ian Bicking* that enables a developer to isolate working environments per project, for easier maintenance of different package versions.

For example, you inherited legacy Django website based on Django 1.1 and Python 2.3, but at the same time you are working on a new project that must be written in Python 2.6. This is my usual case—having more than one required Python version (and related packages) depending on the project I am working on.

virtualenv enables me to easily switch to different environments and have the same package easily reproduced if I need to switch to another machine or to deploy software to a production server (or to a client's workstation).

Getting ready

To install virtualenv, you must have workable installation of Python and pip. Pip is a tool for installing and managing Python packages, and it is a replacement for easy install. We will use pip through most of this book for package management. Pip is easily installed, as root executes the following line in your terminal:

```
# easy_install pip
```

virtualenv by itself is really useful, but with the help of virtualenvwrapper, all

this becomes easy to do and also easy to organize many virtual environments. See all the features at <http://virtualenvwrapper.readthedocs.org/en/latest/#features>.

How to do it...

By performing the following steps you can install the virtualenv and virtualenvwrapper tools: 1. Install virtualenv and virtualenvwrapper:

```
$ sudo pip virtualenv
$ sudo pip virtualenvwrapper
# Create folder to hold all our virtual environments and export the path to it.
$ export VIRTENV=~/.virtualenvs
$ mkdir -p $VIRTENV
# We source (ie. execute) shell script to activate the wrappers $ source /usr/local/bin/virtualenvwrapper.sh
# And create our first virtual environment
$ mkvirtualenv virt1
```

2. You can now install our favorite package inside virt1: **(virt1)user1:~\$ pip install matplotlib** 3. You will probably want to add the following line to your ~/.bashrc file: source /usr/local/bin/virtualenvwrapper.sh

Few useful and most frequently used commands are as follows:

f mkvirtualenv ENV: This creates virtual environment with name ENV and activates it

f workon ENV: This activates the previously created ENV

f deactivate: This gets us out of the current virtual environment

Installing matplotlib on Mac OS X

The easiest way to get matplotlib on Mac OS X is to use prepackaged python distributions such as Enthought Python Distribution (EPD). Just go to the EPD site and download and install the latest stable version for your OS.

In case you are not satisfied with EPD or cannot use it for other reasons such as versions distributed with it, there is a manual (read: harder) way of installing Python, matplotlib, and its dependencies.

Getting ready

We will use the Homebrew project that eases installation of all software that Apple did not install on your OS, including Python and matplotlib. Under the hood, Homebrew is a set of Ruby and Git that automate download and installation. Following these instructions should get the installation working. First, we will install Homebrew, and then Python, followed by tools such as virtualenv, then dependencies for matplotlib (NumPy and SciPy), and finally matplotlib. Hold on, here we go.

How to do it...

1. In your Terminal paste and execute the following command:

ruby <(curl -fsSkL raw.githubusercontent.com/mxcl/homebrew/go)

After the command finishes, try running `brew update` or `brew doctor` to verify that installation is working properly.

2. Next, add the Homebrew directory to your system path, so the packages you install using Homebrew have greater priority than other versions. Open `~/.bash_profile` (or `/Users/[your-user-name]/.bash_profile`) and add the following line to the end of file:

export PATH=/usr/local/bin:\$PATH

3. You will need to restart the terminal so it picks a new path. Installing Python is as easy as firing up another one-liner:

brew install python --framework --universal

This will also install any prerequisites required by Python.

4. Now, you need to update your path (add to the same line):

export PATH=/usr/local/share/python:/usr/local/bin:\$PATH

5. To verify that installation worked, type `python --version` at the command line, you should see 2.7.3 as the version number in the response.

6. You should have pip installed by now. In case it is not installed, use `easy_install` to add pip:

\$ easy_install pip

7. Now, it's easy to install any required package; for example, `virtualenv` and `virtualenvwrapper` are useful:

pip install virtualenv

pip install virtualenvwrapper

8. Next step is what we really wanted to do all along—install matplotlib: **pip install numpy**
brew install gfortran
pip install scipy

Mountain Lion users will need to install the development version of SciPy (0.11) by executing the following line:

pip install -e git+https://github.com/scipy/ scipy#egg=scipy-dev

9. Verify that everything is working. Call Python and execute the following commands: **import numpy**

print numpy.__version__

import scipy

print scipy.__version__

quit()

10. Install matplotlib:

pip install matplotlib

Installing matplotlib on Windows

In this recipe, we will demonstrate how to install Python and start working with matplotlib installation. We assume Python was not previously installed.

Getting ready

There are two ways of installing matplotlib on Windows. The easier way is by installing prepackaged Python environments such as EPD, Anaconda and Python(x,y). This is the suggested way to install Python, especially for beginners.

The second way is to install everything using binaries of precompiled matplotlib and required dependencies. This is more difficult as you have to be careful about the versions of NumPy and SciPy you are installing, as not every version is compatible with the latest version of matplotlib binaries. The advantage in this is that you can even compile your particular versions of matplotlib or any library as to have the latest features, even if they are not provided by authors.

How to do it...

The suggested way of installing free or commercial Python scientific distributions is as easy as following the steps provided on the project's website.

If you just want to start using matplotlib and don't want to be bothered with Python versions and dependencies, you may want to consider using the Enthought Python Distribution (EPD). EPD contains prepackaged libraries required to work with matplotlib and all the required dependencies (SciPy, NumPy, IPython, and more).

As usual, we download Windows Installer (*.exe) that will install all the code we need to start using matplotlib and all recipes from this book.

There is also a free scientific project Python(x,y) (<http://code.google.com/p/pythonxy/>) for Windows 32-bit system that contains all dependencies resolved, and is an easy (and free!) way of installing matplotlib on Windows. Because Python(x,y) is compatible with Python modules installers, it can be easily extended with other Python libraries. No Python installation should be present on the system before installing Python(x,y).

Let me shortly explain how we would install matplotlib using precompiled Python, NumPy, SciPy, and matplotlib binaries. First, we download and install standard Python using official MSI Installer for our platform (x86 or x86-64). After that, download official binaries for NumPy and SciPy and install them first. When you are sure that NumPy and SciPy are properly installed, then we download the latest stable release binary for matplotlib and install it by following the official instructions.

There's more...

Note that many examples are not included in the Windows installer. If you want to try the demos, download the matplotlib source and look in the examples subdirectory.

Installing Python Imaging Library (PIL) for image processing

Python Imaging Library (PIL) enables image processing using Python, has an extensive file format support, and is powerful enough for image processing. Some popular features of PIL are fast access to data, point operations, filtering, image resizing, rotation, and arbitrary affine transforms. For example, the histogram method allows us to get statistics about the images.

PIL can also be used for other purposes, such as batch processing, image archiving, creating thumbnails, conversion between image formats, and printing

images.

PIL reads a large number of formats, while write support is (intentionally) restricted to the most commonly used interchange and presentation formats.

How to do it...

The easiest and most recommended way is to use your platform's package managers. For Debian/Ubuntu use the following commands:

```
$ sudo apt-get build-dep python-imaging
```

```
$ sudo pip install http://effbot.org/downloads/Imaging-1.1.7.tar.gz
```

How it works...

This way we are satisfying all build dependencies using apt-get system but also installing the latest stable release of PIL. Some older versions of Ubuntu usually don't provide the latest releases.

On RedHat/SciLinux:

```
# yum install python-imaging
```

```
# yum install freetype-devel
```

```
# pip install PIL
```

There's more...

There is a good online handbook, specifically, for PIL. You can read it at <http://www.pythonware.com/library/pil/handbook/index.htm>, or download the PDF version from <http://www.pythonware.com/media/data/pil-handbook.pdf>.

There is also a PIL fork, Pillow, whose main aim is to fix installation issues. Pillow can be found at <http://pypi.python.org/pypi/Pillow> and it is easy to install. On Windows, PIL can also be installed using a binary installation file. Install PIL in your Python site-packages by executing .exe from <http://www.pythonware.com/products/pil/>. Now, if you want PIL used in virtual environment, manually copy the PIL.pth file and the PIL directory at C:\Python27\Lib\site-packages to your virtualenv site-packages directory.

Installing a requests module

Most of the data that we need now is available over HTTP or similar protocol, so we need something to get it. Python library requests makes that job easy. Even though Python comes with the urllib2 module for work with remote

resources and supporting HTTP capabilities, it requires a lot of work to get the basic tasks done.

Requests module brings new API that makes the use of web services seamless and pain free. Lot of the HTTP 1.1 stuff is hidden away and exposed only if you need it to behave differently than default.

How to do it...

Using pip is the best way to install requests. Use the following command for the same: **\$ pip install requests**

That's it. This can also be done inside your virtualenv if you don't need requests for every project or want to support different requests versions for each project. Just to get you ahead quickly, here's a small example on how to use requests:

```
import requests
r = requests.get('http://github.com/timeline.json')
print r.content
```

How it works...

We sent the GET HTTP request to a URI at www.github.com that returns a JSON-formatted timeline of activity on GitHub (you can see HTML version of that timeline at <https://github.com/timeline>). After response is successfully read, the `r` object contains content and other properties of the response (response code, cookies set, header metadata, even the request we sent in order to get this response).

Customizing matplotlib's parameters in code

The Library we will use the most throughout this book is matplotlib; it provides the plotting capabilities. Default values for most properties are already set inside the configuration file for matplotlib, called `rcfile`. This recipe describes how to modify matplotlib properties from our application code.

Getting ready

As we already said, matplotlib configuration is read from a configuration file. This file provides a place to set up permanent default values for certain

matplotlib properties, well, for almost everything in matplotlib.

How to do it...

There are two ways to change parameters during code execution: using the dictionary of parameters (rcParams) or calling the matplotlib.rc() command. The former enables us to load already existing dictionary into rcParams, while the latter enables a call to a function using tuple of keyword arguments.

If we want to restore the dynamically changed parameters, we can use matplotlib.rcParamsdefaults() call to restore the standard matplotlib settings.

The following two code samples illustrate previously explained behaviors:
Example for matplotlib.rcParams:

```
import matplotlib as mp
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'
```

Example for the matplotlib.rc() call:
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')

Both examples are semantically the same. In the second sample, we define that all subsequent plots will have lines with line width of 2 points. The last statement of the previous code defines that the color of every line following this statement will be red, unless we override it by local settings. See the following example:

```
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(0.0, 1.0, 0.01)

s = np.sin(2 * np.pi * t)
# make line red
plt.rcParams['lines.color'] = 'r' plt.plot(t,s)

c = np.cos(2 * np.pi * t) # make line thick
plt.rcParams['lines.linewidth'] = '3' plt.plot(t,c)
```

```
plt.show()
```

How it works...

First, we import matplotlib.pyplot and NumPy to allow us to draw sine and cosine graphs. Before plotting the first graph, we explicitly set line color to red using `plt.rcParams['lines.color'] = 'r'`.

Next, we go to the second graph (cosine function), and explicitly set line width to 3 points using `plt.rcParams['lines.linewidth'] = '3'`.
If we want to reset specific settings, we should call `matplotlib.rcdefaults()`.

Customizing matplotlib's parameters per project

This recipe explains where the various configuration files are that matplotlib uses, and why we want to use one or the other. Also, we explain what is in these configuration files.

Getting ready

If you don't want to configure matplotlib as the first step in your code every time you use it (as we did in the previous recipe), this recipe will explain how to have different default configurations of matplotlib for different projects. This way your code will not be cluttered with configuration data and, moreover, you can easily share configuration templates with your co-workers or even among other projects.

How to do it...

If you have a working project that always uses the same settings for certain parameters in matplotlib, you probably don't want to set them every time you want to add a new graph code. Instead, what you want is a permanent file, outside of your code, which sets defaults for matplotlib parameters.

matplotlib supports this via its `matplotlibrc` configuration file that contains most of the changeable properties of matplotlib.

How it works...

There are three different places where this file can reside and its location defines its usage. They are:

f Current working directory: This is where your code runs from. This is the place to customize matplotlib just for your current directory that might contain your current project code. File is named matplotlibrc.

f Per user .matplotlib/matplotlibrc: This is usually in user's \$HOME directory (under Windows, this is your Documents and Settings directory). You can find out where your configuration directory is using the matplotlib.get_configdir() command. Check the next command.

f Per installation configuration file: This is usually in your python site-packages. This is a system-wide configuration, but it will get overwritten every time you reinstall matplotlib; so it is better to use per user configuration file for more persistent customizations. Best usage so far for me was to use this as a default template if I mess up my user's configuration file or if I need fresh configuration to customize for a different project.

The following one-liner will print the location of your configuration directory and can be run from shell.

\$ python -c 'import matplotlib as mpl; print mpl.get_configdir()' The configuration file contains settings for:

f axes: Deals with face and edge color, tick sizes, and grid display.

f backend: Sets the target output: TkAgg and GTKAgg.

f figure: Deals with dpi, edge color, figure size, and subplot settings. f font: Looks at font families, font size, and style settings.

f grid: Deals with grid color and line settings.

f legend: Specifies how legends and text inside will be displayed.

f lines: It checks for line (color, style, width, and so on) and markers settings.

f patch: Patches are graphical objects that fill 2D space, such as polygons and circles;

set linewidth, color, antialiasing, and so on.

f savefig: There are separate settings for saved figures. For example, to make rendered files with a white background.

f text: This looks for text color, how to interpret text (plain versus latex markup) and similar.

f verbose: It checks how much information matplotlib gives during runtime: silent,

helpful, debug, and debug-annoying.

f xticks and yticks: These set the color, size, direction, and labelsizes for major and

minor ticks for x and y axes.

There's more...

If you are interested in more details for every mentioned setting (and some that we did not mention here), the best place to go is the website of matplotlib project where there is up-to-date API documentation. If it doesn't help, user and development lists are always good places to leave questions. See the back of this book for useful online resources.

2

Knowing Your Data

In this chapter we will cover the following recipes:

- f Importing data from CSV
- f Importing data from Microsoft Excel files
- f Importing data from fixed-width datafiles
- f Importing data from tab-delimited files
- f Importing data from a JSON resource
- f Exporting data to JSON, CSV, and Excel
- f Importing data from a database
- f Cleaning up data from outliers
- f Reading files in chunks
- f Reading streaming data sources
- f Importing image data into NumPy arrays
- f Generating controlled random datasets
- f Smoothing the noise in real-world data

Introduction

This chapter covers basics about importing and exporting data from various formats. Also covered are ways of cleaning data, such as normalizing values, adding missing data, live data inspection, and usage of some similar tricks to get data correctly prepared for visualization.

Importing data from CSV

In this recipe we will work with the most common file format that one will encounter in the wild world of data, CSV. It stands for Comma Separated Values, which almost explains all the formatting there is. (There is also a header part of the file, but those values are also comma separated.)

Python has a module called `csv` that supports reading and writing CSV files in various dialects. Dialects are important because there is no standard CSV and

different applications implement CSV in slightly different ways. A file's dialect is almost always recognizable by the first look into the file.

Getting ready

What we need for this recipe is the CSV file itself. We will use sample CSV data that you can download from [ch02-data.csv](#).

We assume that sample datafiles is in the same folder as the code reading it.

How to do it...

The following code example demonstrates how to import data from a CSV file. We will:

1. Open the `ch02-data.csv` file for reading.
2. Read the header first.
3. Read the rest of the rows.
4. In case there is an error, raise an exception.
5. After reading everything, print the header and the rest of the rows.

```
filename = 'ch02-data.csv'
data = []
try:
    with open(filename) as f:

        reader = csv.reader(f)
        header = reader.next()
        data = [row for row in reader]

except csv.Error as e:
    print "Error reading CSV file at line %s: %s" % (reader.line_
num, e)
    sys.exit(-1)
if header:
    print header
    print '======'

for datarow in data: print datarow
```

How it works...

First, we import the `csv` module in order to enable access to required methods. Then, we open the file with data using the `with` compound statement and bind it

to the object `f`. The context manager with statement releases us of care about the closing resource after we are finished manipulating those resources. It is a very handy way of working with resource-like files because it makes sure that the resource is freed (for example, that the file is closed) after the block of code is executed over it.

Then, we use the `csv.reader()` method that returns the reader object, allowing us to iterate over all rows of the read file. Every row is just a list of values and is printed inside the loop.

Reading the first row is somewhat different as it is the header of the file and describes the data in each column. This is not mandatory for CSV files and some files don't have headers, but they are a really nice way of providing minimal metadata about datasets. Sometimes though, you will find separate text or even CSV files that are just used as metadata, describing the format and additional data about the data.

The only way to check what the first line looks like is to open the file and visually inspect it (for example, see the first few lines of the file). This can be done efficiently on Linux using bash commands such as `head` as follows:

\$ head some_file.csv

During iteration of data, we save the first row in header, while we add every other row to the data list.

Should any errors occur during reading, `csv.reader()` will generate an error that we can catch and print the helpful message to the user, in order to help detection of errors.

There's more...

If you want to read about the background and reasoning for the `csv` module, the PEP-defined document *CSV File API* is available at <http://www.python.org/dev/peps/pep-0305/>. If we have larger files that we want to load, it's often better to use well-known libraries, such as NumPy's `loadtxt()`, that cope better with large CSV files.

The basic usage is simple as shown in the following code snippet:

```
import numpy
data = numpy.loadtxt('ch02-data.csv', dtype='string', delimiter=',')
```

Note that we need to define a delimiter to instruct NumPy to separate our data as appropriate. The function `numpy.loadtxt()` is somewhat faster than the similar

function `numpy.genfromtxt()`, but the latter can cope better with missing data, and you are able to provide functions to express what is to be done during the processing of certain columns of loaded datafiles.

Currently, in Python 2.7.x, the `csv` module doesn't support Unicode, and you must explicitly convert the read data into UTF-8 or ASCII printable. The official Python CSV documentation offers good examples on how to resolve data encoding issues.

In Python 3.3 and later versions, Unicode support is default and there are no such issues.

Importing data from Microsoft Excel files

Although Microsoft Excel supports some charting, sometimes you need more flexible and powerful visualization and need to export data from existing spreadsheets into Python for further use.

A common approach to importing data from Excel files is to export data from Excel into CSV-formatted files and use the tools described in the previous recipe to import data using Python from the CSV file. This is a fairly easy process if we have one or two files (and have Microsoft Excel or OpenOffice.org installed), but if we are automating a data pipe for many files (as part of an ongoing data processing effort), we are not in a position to manually convert every Excel file into CSV. So, we need a way to read any Excel file.

Python has decent support for reading and writing Excel files through the project www.python-excel.org. This support is available in the form of different modules for reading and writing, and is platform independent; in other words, we don't have to run on Windows in order to read Excel files.

The Microsoft Excel file format changed over time, and support for different versions is available in different Python libraries. The latest stable version of XLRD is 0.90 at the time of this writing and it has support for reading `.xlsx` files.

Getting ready

First we need to install the required module. For this example, we will use the module `xlrd`. We will use `pip` in our virtual environment.

```
$ mkvirtualenv xlrddexample  
(xlrddexample)$ pip install xlrd
```


After successful installation, use the sample file `ch02-xlsxdata.xlsx`.

How to do it...

The following code example demonstrates how to read a sample dataset from a known Excel file. We will:

1. Open the file workbook.
2. Find the sheet by name.
3. Read the cells using the number of rows (`nrows`) and columns (`ncols`).
4. For demonstration purposes, we only print the read dataset.

```
import xlrd

file = 'ch02-xlsxdata.xlsx'
wb = xlrd.open_workbook(filename=file)
ws = wb.sheet_by_name('Sheet1')
dataset = []

for r in xrange(ws.nrows):
    col = []
    for c in range(ws.ncols):

        col.append(ws.cell(r, c).value)
    dataset.append(col)
from pprint import pprint
pprint(dataset)
```

How it works...

Let us try to explain the simple object model that `xlrd` uses. At the top level, we have a workbook (the Python class `xlrd.book.Book`) that consists of one or more worksheets (`xlrd.sheet.Sheet`), and every sheet has a cell (`xlrd.sheet.Cell`) that we can then read the value from.

We load a workbook from a file using `open_workbook()`, which returns the `xlrd.book.Book` instance that contains all the information about a workbook, such as sheets. We access sheets using `sheet_by_name()`; if we need all sheets, we could use `sheets()`, which returns a list of the `xlrd.sheet.Sheet` instances. The `xlrd.sheet.Sheet` class has a number of columns and rows as attributes that we can use to infer ranges for our loop to access every particular cell inside a worksheet using the method `cell()`. There is an `xlrd.sheet.Cell` class, though it is

not something we want to use directly.

Note that the date is stored as a floating point number and not as a separate data type, but the `xlrd` module is able to inspect the value and try to infer if the data is in fact a date. So we can inspect the cell type for the cell to get the Python date object. The module `xlrd` will return `xlrd.XL_CELL_DATE` as the cell type if the number format string looks like a date. Here is a snippet of code that demonstrates this:

```
from datetime import datetime
from xlrd import open_workbook, xldate_as_tuple
...
cell = sheet.cell(1, 0)
print cell
print cell.value
print cell.ctype
if cell.ctype == xlrd.XL_CELL_DATE:

    date_value = xldate_as_tuple(cell.value, book.datemode)
    print datetime(*date_value)
```

This field still has issues, so please refer to the official documentation and mailing list in case you require extensive work with dates.

There's more...

A neat feature of `xlrd` is its ability to load only parts of the file that are required in the memory. There is an `on_demand` parameter that can be passed the value `True` while calling `open_workbook` so that the worksheet will only be loaded when requested. For example:

```
book = open_workbook('large.xls', on_demand=True)
```

We didn't mention writing Excel files in this section, partly because there will be a separate recipe for that and partly because there is a different module for that —`xlwt`. You will read more about it in the *Exporting data to JSON, CSV, and Excel* recipe in this chapter. If you need specific usage that was not covered with the module and examples explained earlier, here is a list of other Python modules on PyPi that might help you out with spreadsheets:
<http://pypi.python.org/pypi?:action=browse&c=377>.

Importing data from fixed-width datafiles

Logfiles from events and time series datafiles are common sources for data visualizations. Sometimes, we can read them using CSV dialect for tab-separated data, but sometimes they are not separated by any specific character. Instead, fields are of fixed widths and we can infer the format to match and extract data.

One way to approach this is to read a file line by line and then use string manipulation functions to split a string into separate parts. This approach seems straightforward, and if performance is not an issue, should be tried first.

If performance is more important or the file to parse is large (hundreds of megabytes), using the Python module `struct` (<http://docs.python.org/library/struct.html>) can speed us up as the module is implemented in C rather than in Python.

Getting ready

As the module `struct` is part of the Python Standard Library, we don't need to install any additional software to implement this recipe.

How to do it...

We will use a pregenerated dataset with a million rows of fixed-width records. Here's what sample data looks like:

```
...
207152670 3984356804116 9532
427053180 1466959270421 5338
316700885 9726131532544 4920
138359697 3286515244210 7400
476953136 0921567802830 4214
213420370 6459362591178 0546
...
```

This dataset is generated using code that can be found in the repository for this chapter, `ch02-generate_f_data.py`.

Now we can read the data. We can use the following code sample. We will:

1. Define the datafile to read.
2. Define the mask for how to read the data.

3. Read line by line using the mask to unpack each line into separate data fields.
4. Print each line as separate fields.

```
import struct
import string
```

```
datafile = 'ch02-fixed-width-1M.data'
```

```
# this is where we define how to # understand line of data from the file
mask='9s14s5s'
```

```
with open(datafile, 'r') as f:
    for line in f:
        fields = struct.Struct(mask).unpack_from(line)
        print 'fields: ', [field.strip() for field in fields]
```

How it works...

We define our format mask according to what we have previously seen in the datafile. To see the file, we could have used Linux shell commands, such as `head` or `more`, or something similar.

String formats are used to define the expected layout of the data to extract. We use format characters to define what type of data we expect. So if the mask is defined as `9s15s5s`, we can read that as "a string of width nine characters, followed by a string width of 15 characters, further followed by a string of five characters."

In general, `c` defines the character (the `char` type in C) or a string of length 1, `s` defines a string (the `char[]` type in C), `d` defines a float (the `double` type in C), and so on. The complete table is available on the official Python website at <http://docs.python.org/library/struct.html#format-characters>.

We then read the file line by line and extract (the `unpack_from` method) the line according to the specified format. Because we might have extraneous spaces before (or after) our fields, we use `strip()` to strip every extracted field.

For unpacking, we used the object-oriented (OO) approach using the `struct.Struct` class, but we could have as well used the non-object approach where the line would be: `fields = struct.unpack_from(mask, line)`

The only difference is the usage of pattern. If we are to perform more processing using the same formatting mask, the OO approach saves us from stating that format in every call. Also, it gives us the ability to inherit the `struct.Struct` class in future, extending or providing additional functionality for specific needs.

Importing data from tab-delimited files

Another very common format of flat datafile is the tab-delimited file. This can also come from an Excel export but can be the output of some custom software we must get our input from.

The good thing is that usually this format can be read in almost the same way as CSV files, as the Python module `csv` supports so-called dialects that enable us to use the same principles to read variations of similar file formats—one of them being the tab delimited format.

Getting ready

We are already able to read CSV files. If not, please refer the *Importing data from CSV* recipe first.

How to do it...

We will re-use the code from the *Importing data from CSV* recipe, where all we need to change is the dialect we are using.

```
import csv
filename = 'ch02-data.tab'
data = []
try:
    with open(filename) as f:
        reader = csv.reader(f, dialect=csv.excel_tab)
        header = reader.next()
        data = [row for row in reader]

except csv.Error as e:
    print "Error reading CSV file at line %s: %s" % (reader.line_num,
    e)
    sys.exit(-1)

if header:
    print header
    print '====='
```

```
for datarow in data: print datarow
```

How it works...

The dialect-based approach is very similar to what we already did in the *Importing data from CSV* recipe, except for the one line where we instantiate the csv reader object, giving it the parameter dialect and specifying the 'excel_tab' dialect that we want.

There's more...

A CSV-based approach will not work if the data is "dirty", that is, if there are certain lines not ending with just a new line character but that instead have additional `\t` (tab) markers. So we need to clean special lines separately before splitting them. The sample "dirty" tab-delimited file can be found in `ch02-data-dirty.tab`. The following code sample cleans data as it reads it:

```
datafile = 'ch02-data-dirty.tab'

with open(datafile, 'r') as f:
    for line in f:
        # remove next comment to see line before cleanup # print 'DIRTY: ',
        line.split('\t')

        # we remove any space in line start or end line = line.strip()
        # now we split the line by tab delimiter print line.split('\t')
```

We see also that there is another approach—using the `split('\t')` function.

The advantage of using the csv module approach over `split()` sometimes is that we can re-use the same code for reading by just changing the dialect and detecting the dialect with the file extension (`.csv` and `.tab`) or some other method (for example, using the `csv.Sniffer` class).

Importing data from a JSON resource

This recipe will show us how we can read the JSON data format. Moreover, we will be using a remote resource in this recipe. It will add a tiny level of complexity to the recipe, but it will make it much more useful because, in real

life, we will encounter more remote resources than local.

JavaScript Object Notation (JSON) is widely used as a platform-independent format to exchange data between systems or applications.

A resource, in this context, is anything we can read, be it a file or a URL endpoint (which can be the output of a remote process/program or just a remote static file). In short, we don't care who produced a resource and how; we just need it to be in a known format, such as JSON.

Getting ready

In order to get started with this recipe, we need the requests module installed and importable (in PYTHONPATH) in our virtual environment. We have installed this module in *Chapter 1, Preparing Your Working Environment*.

We also need Internet connectivity as we will be reading a remote resource.

How to do it...

The following code sample performs reading and parsing of the recent activities timeline from the GitHub (<http://github.com>) site. We will perform the following steps for this:

1. Define the GitHub URL to read the JSON format.
2. Get the contents from the URL using the requests module.
3. Read the content as JSON.
4. For each entry in the JSON object, read the URL value for each repository.

```
import requests
```

```
url = 'https://github.com/timeline.json'
```

```
r = requests.get(url)
```

```
json_obj = r.json()
```

```
repos = set()
```

```
for entry in json_obj:
```

```
try:
```

```
    repos.add(entry['repository']['url'])
```

```
except KeyError as e:
```

```
    print "No key %s. Skipping..." % (e)
```

```
from pprint import pprint
pprint(repos)
```

How it works...

First, we use the `requests` module to fetch a remote resource. This is very straightforward as the `requests` module offers a simple API to define HTTP verbs, so we just need to issue one `get()` method call. This method retrieves data and request metadata and wraps it in the `Response` object so we can inspect it. For this recipe, we are only interested in the `Response.json()` method, which automatically reads content (available at `Response.content`) and parses it as JSON and loads it into the JSON object.

Now that we have the JSON object, we can process data. In order to do that, we need to understand what data looks like. We can achieve that understanding by opening the JSON resource using our favorite web browser or command-line tool such as `wget` or `curl`.

Another way is to fetch data from IPython and inspect it interactively. We can achieve that by running our program from IPython (using `%run program_name.py`). After execution, we are left with all variables that the program produced. List them all using `%who` or `%whos`.

Whatever method we use, we gain knowledge about the structure of the JSON data and the ability to see what parts of that structure we are interested in.

The JSON object is basically just a Python dictionary (or if more complex, a dictionary of dictionaries) and we can access parts of it using a well-known, key-based notation. We get our list of URLs of recently updated repositories referencing `entry['repository']['url']`.

`entry['repository']['url']` matches this section in the actual JSON file:

```
...
"repository" : {
...
"url" : "https://github.com/ipython/ipython", ...
},
...
```

We can now see how nested structures correspond to multidimensional key indexes in Python code.

There's more...

The JSON format (specified by RFC 4627; refer to <http://tools.ietf.org/html/rfc4627.html>) became very popular recently, as it is more human readable than XML and is also less verbose. Hence, it's lighter in terms of the syntaxes required to transfer data. It is very popular in the web application domain as it is native to JavaScript, the language used for most of today's rich Internet applications.

The Python JSON module has more capabilities than we have displayed here; for example, we could specialize the basic JSONEncoder/JSONDecoder class to transform our Python data into JSON format. The classical example uses this approach to JSON-ify the Python built-in type for complex numbers.

For simple customization, we don't have to subclass the JSONDecoder/JSONEncoder class as some of the parameters can solve our problems.

For example, `json.loads()` will parse a float as the Python type float, and most of the time it will be right. Sometimes, however, the float value in the JSON file represents a price value, and this is better represented as a decimal. We can instruct the json parser to parse floats as decimal. For example, we have this JSON string:

```
jstring = '{"name":"prod1","price":12.50}'
```

This is followed by these two lines of code:

```
from decimal import Decimal
```

```
json.loads(jstring, parse_float=Decimal)
```

The preceding two lines of code will generate this output:

```
{u'name': u'prod1', u'price': Decimal('12.50')}
```

Exporting data to JSON, CSV, and Excel

While, as producers of data visualization, we are mostly using other people's data; importing and reading data are major activities. We do need to write or export data that we produced or processed, whether it is for our or others' current or future use.

We will demonstrate how to use the previously mentioned Python modules to import, export, and write data to various formats such as JSON, CSV, and

XLSX.

For demonstration purposes, we are using the pregenerated dataset from the *Importing data from fixed-width datafiles* recipe.

Getting ready

For the Excel writing part, we will need to install the xlwt module (inside our virtual environment) by executing the following command:

\$ pip install xlwt

How to do it...

We will present one code sample that contains all the formats that we want to demonstrate: CSV, JSON, and XLSX. The main part of the program accepts the input and calls appropriate functions to transform data. We will walk through separate sections of code, explaining its purpose.

1. Import the required modules.

```
import os
```

```
import sys
```

```
import argparse
```

```
try: import cStringIO as StringIO
```

```
except:
```

```
import StringIO
```

```
import struct
```

```
import json
```

```
import csv
```

2. Then, define the appropriate functions for reading and writing data. def

```
import_data(import_file):
```

```
'''
```

```
Imports data from import_file.
```

```
Expects to find fixed width row
```

```
Sample row: 161322597 0386544351896 0042
```

```
'''
```

```
mask = '9s14s5s'
```

```
data = []
```

```
with open(import_file, 'r') as f:
```

```
for line in f:
```

```
# unpack line to tuple
```

```
fields = struct.Struct(mask).unpack_from(line) # strip any whitespace for each field
# pack everything in a list and add to full dataset data.append(list([f.strip() for f
in fields]))
```

```
return data
```

```
def write_data(data, export_format):
    """Dispatches call to a specific transformer and returns data
    set.
    Exception is xlsx where we have to save data in a file. """
    if export_format == 'csv':
        return write_csv(data)
    elif export_format == 'json':
        return write_json(data)
    elif export_format == 'xlsx':
        return write_xlsx(data)
    else:
        raise Exception("Illegal format defined")
```

3. We separately specify separate implementation for each data format (CSV, JSON, and XLSX).

```
def write_csv(data):
```

```
    """Transforms data into csv. Returns csv as string. """
    # Using this to simulate file IO,
    # as csv can only write to files.
    f = StringIO.StringIO()
    writer = csv.writer(f)
    for row in data:
```

```
        writer.writerow(row)
    # Get the content of the file-like object
    return f.getvalue()
```

```
def write_json(data):
    """Transforms data into json. Very straightforward. """
    j = json.dumps(data)
    return j
```

```

def write_xlsx(data):
    """Writes data into xlsx file.

    """
    from xlwt import Workbook
    book = Workbook()
    sheet1 = book.add_sheet("Sheet 1")
    row = 0
    for line in data:

        col = 0
        for datum in line:
            print datum
            sheet1.write(row, col, datum)
            col += 1
        row += 1
        # We have hard limit here of 65535 rows # that we are able to save in
        # spreadsheet.
        if row > 65535:
            print >> sys.stderr, "Hit limit of # of rows in one sheet (65535)."
            break
        # XLS is special case where we have to
        # save the file and just return 0
        f = StringIO.StringIO()
        book.save(f)
        return f.getvalue()

```

4. Finally, we have the main code entry point, where we parse argument-like files from the command line to import data and export it to the required format.

```

if __name__ == '__main__':
    # parse input arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("import_file", help="Path to a fixed-width
    data file.")
    parser.add_argument("export_format", help="Export format:
    json, csv, xlsx.")
    args = parser.parse_args()

```

```

if args.import_file is None:
    print >> sys.stderr, "You must specify path to import from."
    sys.exit(1)
if args.export_format not in ('csv','json','xlsx'): print >> sys.stderr, "You must
provide valid export file format."
sys.exit(1)
# verify given path is accessible file
if not os.path.isfile(args.import_file):
    print >> sys.stderr, "Given path is not a file: %s" % args.import_file
    sys.exit(1)
# read from formatted fixed-width file data = import_data(args.import_file)

# export data to specified format
# to make this Unix-like pipe-able # we just print to stdout
print write_data(data, args.export_format)

```

How it works...

In one broad sentence, we import the fixed-width dataset (defined in the *Importing data from fixed-width datafiles* recipe) and then export that to stdout so we can catch that in a file or as input to another program.

We call out the programmer from the command line giving two mandatory arguments: the input filename and the export data format (JSON, CSV, and XLSX).

If we successfully parse those arguments, we dispatch the input file reading to a function `import_data()`, which returns the Python data structure (list of lists) that we can easily manipulate to get to the appropriate export format.

We route our request inside the `write_data()` function, where we just forward a call to the appropriate function (`write_csv()`, for example).

For CSV, we obtain the `csv.writer()` instance that we use to write every line of data we iterate over.

We just return the given string as we will redirect this output from our program to another program (or just to cat in a file).

The JSON export is not required for this example, as the `json` module provides us with the `dump()` method that happily reads our Python structure. Just as for CSV, we simply return and dump this output to stdout.

The Excel export requires more code, as we need to create a more complex model of the Excel workbook and worksheet(s) that will hold the data. This activity is followed by a similar iterative approach. We have two loops, the outer one goes over every line in the source dataset iterated, while the inner one iterates over every field in the given line.

After all this, we save the Book instance into a file-like stream that we can return to stdout and use it both in read files and the files consumed by the web service.

There's more...

This, of course, is just a small set of possible data formats that we could be exporting to. It is fairly easy to modify the behavior. Basically, two places need changes: the import and export functions. The function for import needs to change if we want to import a new kind of data source.

If we want to add a new export format, we need to first add functions that will return a stream of formatted data. Then, we need to update the `write_data()` function to add the new `elif` branch to have it call our new `write_*` function. One thing we could also do is make this a Python package, so we can re-use it over more projects. In that case, we would like to make import more flexible and maybe add some more configuration features for import.

Importing data from a database

Very often, our work on data analysis and visualization is at the consumer end of the data pipeline. We most often use the already produced data, rather than producing the data ourselves. A modern application, for example, holds different datasets inside relational databases (or other databases), and we use these databases and produce beautiful graphs.

This recipe will show you how to use SQL drivers from Python to access data.

We will demonstrate this recipe using a SQLite database because it requires the least effort to setup, but the interface is similar to most other SQL-based database engines (MySQL and PostgreSQL). There are, however, differences in the SQL dialect that those database engines support. This example uses simple SQL language and should be reproducible on most common SQL database engines.

Getting ready

To be able to execute this recipe, we need to install the SQLite library.

\$ sudo apt-get install sqlite3

Python support for SQLite is there by default, so we don't need to install anything Python related. Just fire the following code snippet in IPython to verify that everything is there:

```
import sqlite3
sqlite3.version
sqlite3.sqlite_version
```

We get the output similar to this:

In [1]: import sqlite3

In [2]: sqlite3.version Out[2]: '2.6.0'

In [3]: sqlite3.sqlite_version Out[3]: '3.6.22'

Here, `sqlite3.version` gets us the version of the Python `sqlite3` module, and `sqlite3.sqlite_version` returns the system SQLite library version.

How to do it...

To be able to read from the database, we need to:

1. Connect to the database engine (or the file in the case of SQLite).
2. Run the query against the selected tables.
3. Read the result returned from the database engine.

I will not try to teach SQL here, as there are many books on that particular topic. But just for clarity, we will explain the SQL query in this code sample:

SELECT ID, Name, Population FROM City ORDER BY Population DESC LIMIT 1000

ID , Name, and Population are columns (fields) of the table City from which we select data. ORDER BY tells the database engine to sort our data by the Population column, and DESC means descending order. LIMIT allows us to get just the first 1,000 records found.

For this example, we will use the `world.sql` example table, which holds the world's city names and populations. This table has more than 5,000 entries. This is how our table looks:

1	ID	Name	Population
2			
3	1024	Mumbai (Bombay)	10500000
4	2331	Seoul	9981619
5	206	São Paulo	9968485
6	1890	Shanghai	9696300
7	939	Jakarta	9604900
8	2822	Karachi	9269265
9	3357	Istanbul	8787958
10	2515	Ciudad de México	8591309
11	3580	Moscow	8389200
12	3793	New York	8008278
13	1532	Tokyo	7980230
14	1891	Peking	7472000
15	456	London	7285000
16	1025	Delhi	7206704
17	608	Cairo	6789479
18	1380	Teheran	6758845
19	2890	Lima	6464693
20	1892	Chongqing	6351600
21	3320	Bangkok	6320174
22	2257	Santafé de Bogotá	6260862

First we need to import

this SQL file into the SQLite database. Here's how to do it:

```
import sqlite3
```

```
import sys
```

```
if len(sys.argv) < 2:
```

```
    print "Error: You must supply at least SQL script." print "Usage: %s table.db
```

```
./sql-dump.sql" % (sys.argv[0]) sys.exit(1)
```

```
script_path = sys.argv[1]
```

```
if len(sys.argv) == 3:
```

```
    db = sys.argv[2]
```

```
else:
```

```
    # if DB is not defined # create memory database db = ":memory:"
```

```
try: con = sqlite3.connect(db)
```

```
with con:
```

```
    cur = con.cursor()
```

```
    with open(script_path,'rb') as f: cur.executescript(f.read()) except sqlite3.Error as
```

```
err:
```

```
    print "Error occurred: %s" % err
```


This reads the SQL file and executes the SQL statements against the opened SQLite db file. If we don't specify the filename, SQLite creates the database in the memory. The statements are then executed line by line.

If we encounter any errors, we catch exceptions and print the error message to the user. After we have imported data into the database, we are able to query the data and do some processing. Here is the code to read the data from the database file:

```
import sqlite3
import sys

if len(sys.argv) != 2:
    print "Please specify database file." sys.exit(1)

db = sys.argv[1]

try: con = sqlite3.connect(db)
with con:

    cur = con.cursor()
    query = 'SELECT ID, Name, Population FROM City ORDER BY

Population DESC LIMIT 1000'
    con.text_factory = str cur.execute(query)

    resultset = cur.fetchall()
    # extract column names

    col_names = [cn[0] for cn in cur.description] print "%10s %30s %10s" %
tuple(col_names) print "="*(10+1+30+1+10)

    for row in resultset:
        print "%10s %30s %10s" % row except sqlite3.Error as err:
        print "[ERROR]:", err
```

How it works...

First, we verify that the user has provided the database filepath. This is just a quick check that we can proceed with the rest of the code.

Then, we try to connect to the database; if that fails, we catch `sqlite3.Error` and print it to the user.

If the connection is successful, we obtain a cursor using `con.cursor()`. A cursor is an iterator-like structure that enables us to traverse records of the result set returned from a database.

We define a query that we execute over the connection and we fetch the result set using `cur.fetchall()`. Had we expected just one result, we would have used just `fetchone()`.

List comprehension over `cur.description` allows us to obtain column names. `description` is a read-only attribute and returns more than we need for just column names, so we just fetch the first item from every column's 7-item tuple.

We then use simple string formatting to print the header of our table with column names. After that, we iterate over resultset and print every row in a similar manner.

There's more...

Databases are the most common sources of data today. We could not present everything in this short recipe, but we can suggest where to look further. The official Python documentation is the first place to look for an explanation about how to work with databases. The most common databases are open source databases—such as MySQL, PostgreSQL, and SQLite—and on the other end of the spectrum, there are enterprise database systems, such as MS SQL, Oracle, and Sybase. Mostly Python has support for them and the interface is abstracted always, so you don't have to change your program if your underlying database changes, but some tweaks may be required. It depends on whether you have used the specifics of a particular database system. For example, Oracle supports a specific language PL/SQL that is not standard SQL, and some things will not work if your database changes from Oracle to MS SQL. Similarly, SQLite does not support specifics from MySQL data types or database engine types (MyISAM and InnoDB). Those things can be annoying, but having your code rely on standard SQL (<http://en.wikipedia.org/wiki/SQL:2011>) will make your code portable from one database system to another.

Cleaning up data from outliers

This recipe describes how to deal with datasets coming from the real world and how to clean them before doing any visualization.

We will present a few techniques, different in essence but with the same goal,

which is to get the data cleaned.

However, cleaning should not be fully automatic. We need to understand the data as given and be able to understand what the outliers are and what the data points represent before we apply any of the robust modern algorithms made to clean the data. This is not something that can be defined in a recipe because it relies on vast areas such as statistics, knowledge of the domain, and a good eye (and then some luck).

Getting ready

We will use the standard Python modules we already know about, so no additional installation is required.

In this recipe, I will introduce a new term, MAD. Median absolute deviation (MAD) in statistics represents a measure of the variability of a univariate (possessing one variable) sample of quantitative data. It is a measure of statistical dispersion. It falls into a group of robust statistics, such that it is more resilient to outliers.

How to do it...

Here's one example that shows how to use MAD to detect outliers in our data. We will perform the following steps for this:

1. Generate normally distributed random data.
2. Add in a few outliers.
3. Use the function `is_outlier()` to detect the outliers.
4. Plot both the datasets (x and filtered) to see the difference.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def is_outlier(points, threshold=3.5):
    """
```

Returns a boolean array with True if points are outliers and

False
otherwise.

Data points with a modified z-score greater than this # value will be classified as outliers.

```

"""
# transform into vector
if len(points.shape) == 1:

    points = points[:,None]
# compute median value
median = np.median(points, axis=0)

# compute diff sums along the axis
diff = np.sum((points - median)**2, axis=-1) diff = np.sqrt(diff)
# compute MAD
med_abs_deviation = np.median(diff)

# compute modified Z-score

# http://www.itl.nist.gov/div898/handbook/eda/section4/eda43.htm#Iglewicz
modified_z_score = 0.6745 * diff / med_abs_deviation

# return a mask for each outlier return modified_z_score > threshold
# Random data
x = np.random.random(100)
# histogram buckets buckets = 50
# Add in a few outliers
x = np.r_[x, -49, 95, 100, -100]

# Keep valid data points
# Note here that
# "~" is logical NOT on boolean numpy arrays filtered = x[~is_outlier(x)]
# plot histograms plt.figure()

plt.subplot(211)
plt.hist(x, buckets) plt.xlabel('Raw')

plt.subplot(212)
plt.hist(filtered, buckets) plt.xlabel('Cleaned')

plt.show()

```

Note that, in NumPy, the `~` operator is overloaded to operate as a logical operator, not on Boolean arrays. For example, suppose we fire up IPython in pylab mode.

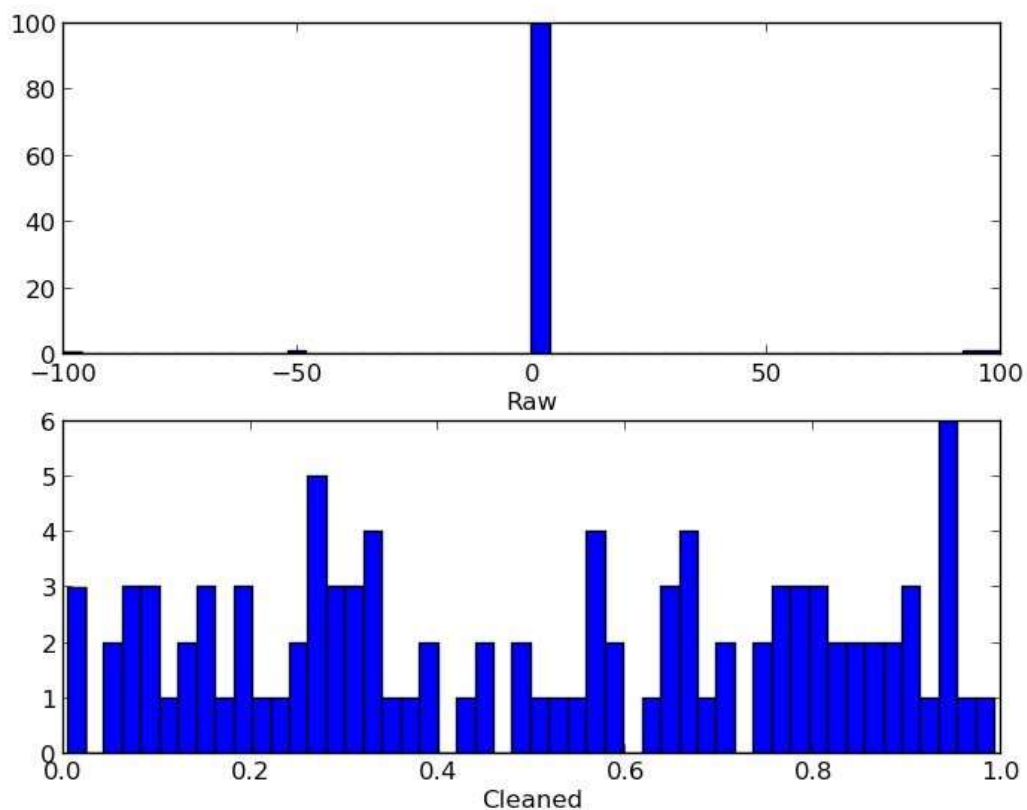
```
$ ipython --pylab
```

We get:

```
In [1]: ~numpy.array(False)
```

```
Out[1]: True
```

We should see two distinct histograms, the first one showing almost nothing—except in one bucket where the biggest outlier is—and the second histogram showing diversified data buckets because we removed outliers.



Another way to identify outliers is to visually inspect your data. In order to do so, we could create scatter plots, where we could easily spot values that are out of the central swarm. We can also make a box plot, which will display the median, quartiles above and below the median, and points that are distant to this box.

The box extends from the lower to upper quartile values of the data, with a line

at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Here's an example to demonstrate that:

```
from pylab import *
```

```
# fake up some data spread= rand(50) * 100
center = ones(25) * 50
```

```
# generate some outliers high and low flier_high = rand(10) * 100 + 100
flier_low = rand(10) * -100
```

```
# merge generated data set
data = concatenate((spread, center, flier_high, flier_low), 0)
```

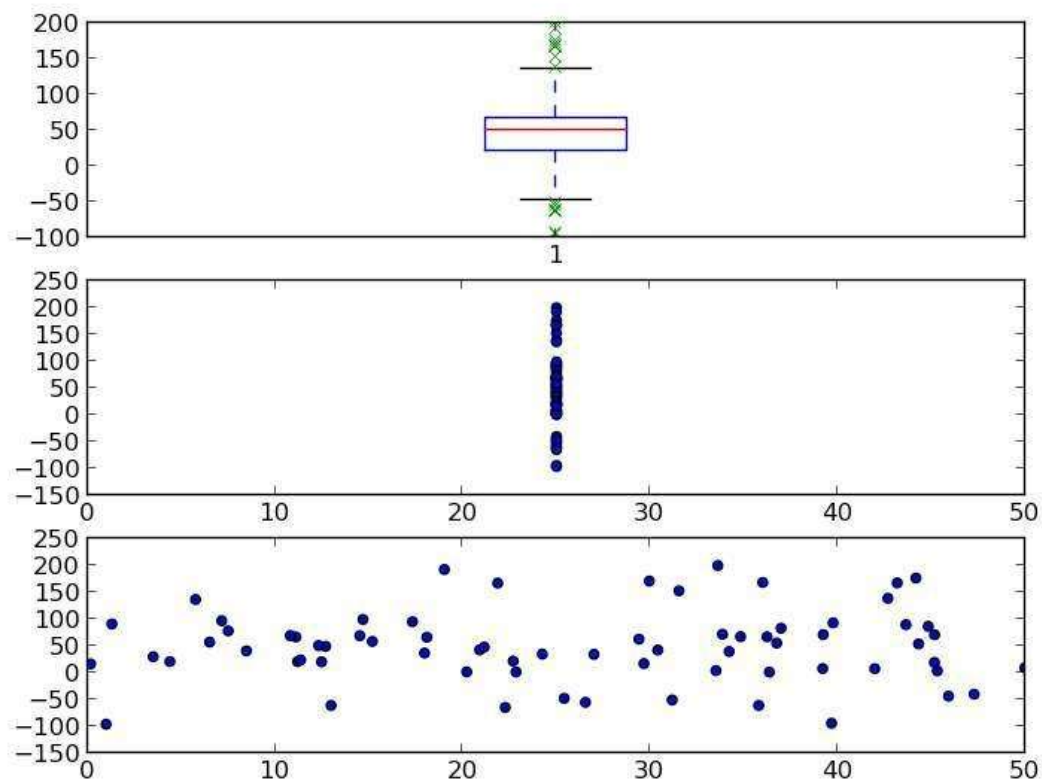
```
subplot(311)
# basic plot
# 'gx' defining the outlier plotting properties boxplot(data, 0, 'gx')
```

```
# compare this with similar scatter plot
subplot(312)
spread_1 = concatenate((spread, flier_high, flier_low), 0) center_1 = ones(70) *
25
scatter(center_1, spread_1)
xlim([0, 50])
```

```
# and with another that is more appropriate for # scatter plot
subplot(313)
center_2 = rand(70) * 50
scatter(center_2, spread_1)
xlim([0, 50])
```

```
show()
```

We can then see x-shaped markers representing outliers:



We can also see that the second plot showing a similar dataset in the scatter plot is not very intuitive because the x axis has all the values at 25 and we don't really distinguish between inliers and outliers.

The third plot, where we generated values on the x axis to be spread across the range from 0 to 50, gives us more visibility of the different values and we can see what values are outliers in terms of the y axis.

In the following code example, we see how the same data (in this example, uniformly distributed) can display itself very differently and sometimes deceptively convey some information that is not really true:

```
# generate uniform data points
x = 1e6*rand(1000)
y = rand(1000)

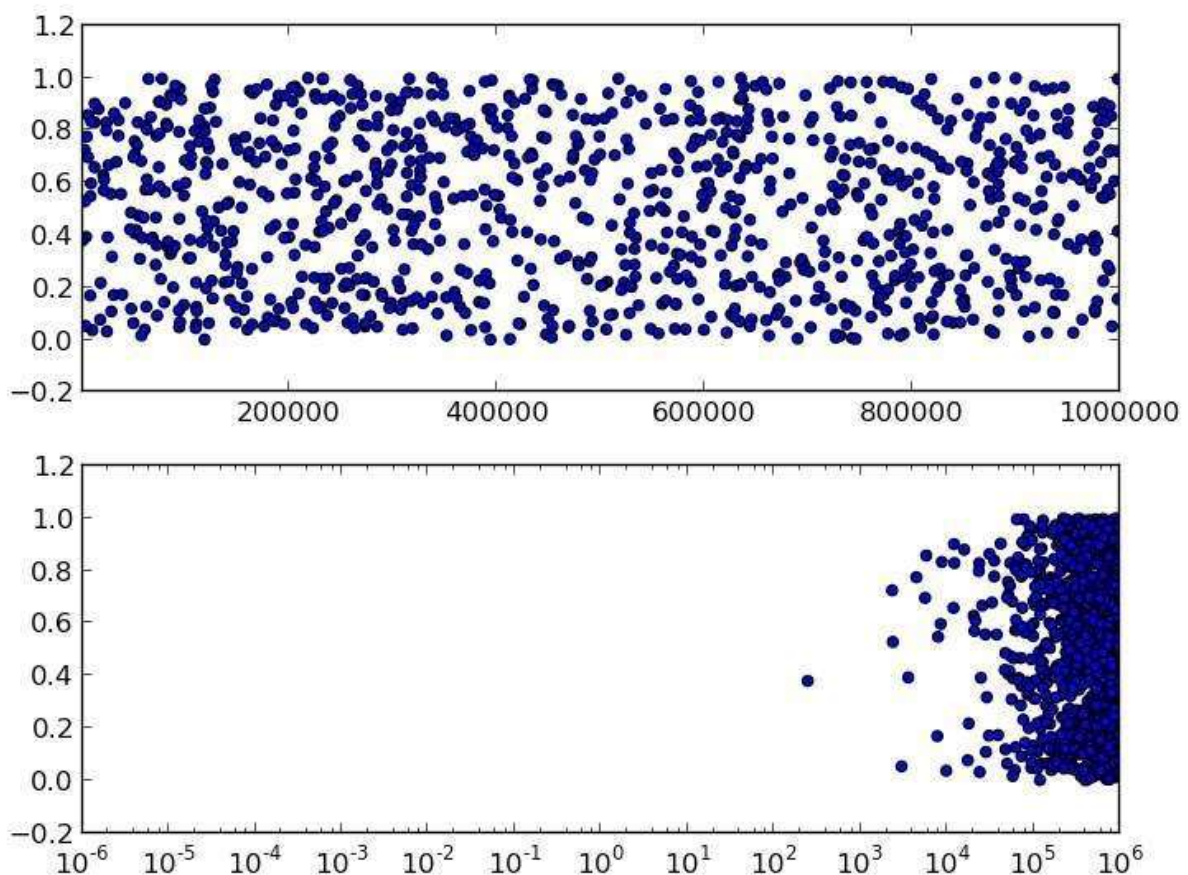
figure()
```

```
# crate first subplot subplot(211)
# make scatter plot scatter(x, y)
# limit x axis
xlim(1e-6, 1e6)

# crate second subplot
subplot(212)
# make scatter plot
scatter(x,y)
# but make x axis logarithmic xscale('log')
# set same x axis limit
xlim(1e-6, 1e6)
```

show()

This is the resulting output:



What if we have a dataset with missing values? We can use NumPy loaders to compensate for missing values, or we can write code to replace existing values

with the ones we need for further use.

Say we want to illustrate some dataset over the geographical map of USA and may have values for state names that are not consistent in the dataset. For example, we have values OH, Ohio, OHIO, US-OH, and OH-USA all representing the state of Ohio in the USA. What we must do in this situation is inspect the dataset manually, either loading it in a spreadsheet processor such as Microsoft Excel or OpenOffice.org Calc. Sometimes, it is easy enough to just print all the lines using Python. If the file is CSV or CSV-like, we can open it with any text editor and inspect the data directly.

After we have concluded about what is in the data, we can write Python code to group those similar values and replace them with the one value that is going to make further processing consistent. The usual way of doing this is to read in lines of the file using `readlines()` and use standard Python string manipulation functions to perform manipulations.

There's more...

There are special products, both commercial and non-commercial (such as OpenRefine—<https://github.com/OpenRefine>), that provide some automation around transformation on "dirty" live datasets.

Still, manual work is involved, depending on how noisy the data is and how great our understanding of that data is.

If you want to find out more about cleaning outliers and cleaning of data in general, look for statistical models and the sampling theory.

Reading files in chunks

Python is very good at handling reading and writing files or file-like objects. For example, if you try to load big files, say a few hundred MB, assuming you have a modern machine with at least 2 GB of RAM, Python will be able to handle it without any issue. It will not try to load everything at once, but play smart and load it as needed.

So even with decent file sizes, doing something as simple as the following code will work straight out of the box:

with `open('/tmp/my_big_file', 'r')` as `bigfile`:

```
for line in bigfile:  
# line based operation, like 'print line'
```

But if we want to jump to a particular place in the file or do other nonsequential reading, we will need to use the handcrafted approach and use IO functions such as seek(), tell(), read(), and next() that allow enough flexibility for most users. Most of these functions are just bindings to C implementations (and are OS-specific), so they are fast, but their behavior can vary based on the OS we are running.

How to do it...

Depending on what our aim is, processing large files can sometimes be managed in chunks. For example, you could read 1,000 lines and process them using Python standard iterator-based approaches.

```
import sys  
filename = sys.argv[1] # must pass valid file name  
  
with open(filename, 'rb') as hugefile:  
    chunksize = 1000  
    readable = "  
# if you want to stop after certain number of blocks # put condition in the while  
while hugefile:  
  
    # if you want to start not from 1st byte  
    # do a hugefile.seek(skipbytes) to skip  
    # skipbytes of bytes from the file start  
    start = hugefile.tell()  
    print "starting at:", start  
    file_block = " # holds chunk_size of lines for _ in xrange(start, start +  
    chunksize):  
  
    line = hugefile.next()  
    file_block = file_block + line  
    print 'file_block', type(file_block), file_block  
  
    readable = readable + file_block  
    # tell where are we in file  
    # file IO is usually buffered so tell()
```

```
# will not be precise for every read.
stop = hugefile.tell()
print 'readable', type(readable), readable print 'reading bytes from %s to %s' %
(start, stop) print 'read bytes total:', len(readable)

# if you want to pause read between chunks # uncomment following line
#raw_input()
```

We call this code from the Python command-line interpreter, giving the filename path as the first parameter.

\$ python ch02-chunk-read.py myhugefile.dat

How it works...

We want to be able to read blocks of lines for processing without reading the whole file in the memory.

We open the file and read in lines in the inner for loop. The way we move through the file is by calling `next()` on the file object. This function reads the line from the file and moves the file pointer to the next line. We append lines in the `file_block` variable during the loop execution. In order to simplify the example code, we don't do any processing but just add `file_block` to complete the output variable `readable`.

We do some printing during execution just to illustrate the current state of certain variables. The last comment line in the while loop, `raw_input()`, can be uncommented and we can pause the execution and read the printed lines above it.

There's more...

This recipe is, of course, just one of the possible approaches to reading large (huge) files. Other approaches could include specific Python or C libraries, but they all depend on what it is that we aim to do with data and how we want to process it.

Parallel approaches such as the MapReduce paradigm have become very popular recently as we get more processing power and memory for a low price.

Multiprocessing is also a feasible approach sometimes, as Python has good library support for creating and managing threads with several libraries such as `multiprocessing`, `threading`, and `thread`.

If processing huge files is a repeated process for a project, we suggest building your data pipeline so that every time you need data ready in a specific format on the output end, you don't have to go to the source and do it manually.

Reading streaming data sources

What if the data that is coming from the source is continuous? What if we need to read continuous data? This recipe will demonstrate a simple solution that will work for many common real-life scenarios, though it is not universal and you will need to modify it if you hit a special case in your application.

How to do it...

In this recipe, we will show you how to read an always-changing file and print the output. We will use the common Python module to accomplish this.

```
import time
import os
import sys

if len(sys.argv) != 2:
    print >> sys.stderr, "Please specify filename to read"
    filename = sys.argv[1]
if not os.path.isfile(filename):
    print >> sys.stderr, "Given file: \"%s\" is not a file" % filename

with open(filename, 'r') as f:
    # Move to the end of file
    filesize = os.stat(filename)[6]
    f.seek(filesize)

    # endlessly loop
    while True:
        where = f.tell()
        # try reading a line
        line = f.readline()
        # if empty, go back
        if not line:
            time.sleep(1)
            f.seek(where)
```

else:

, at the end prevents print to add newline, as readline()

already read that. print line,

How it works...

The core of the code is inside the `while True:` loop. This loop never stops (unless we interrupt it by pressing *Ctrl + C* on our keyboard). We first move to the end of the file we are reading and then we try to read a line. If there is no line, that means nothing was added to the file after we checked it using `seek()`. So, we sleep for one second and then try again.

If there is a nonempty line, we print that out and suppress the new line character.

There's more...

We might want to read the last n lines. We could do that by going almost to the end of the file. We could go there by looking for the file, that is, with `file.seek(filesize - N * avg_line_len)`. Here, `avg_line_len` should be the approximation of average line length in that file (approximately 1,024). Then, we could use `readlines()` from that point to read line, and then print just `[-N]` lines from that list.

The idea from this example can be used for various solutions. For example, the input has to be a file-like object or a remote HTTP-accessible resource. Thus, one can read the input from a remote service and continuously parse it and update live charts, for example, or update the intermediate queue, buffer, or database.

One particular module is very useful for stream handling—`io`. It is in Python from Version 2.6, is built as a replacement for the `file` module, and is a default interface in Python 3.x.

In some more complex data pipelines, we will need to enable some sort of message queues, where our incoming continuous data will have to be queued for some time before we are able to accept it. This enables us, as consumers of the data, to be able to pause processing if we are overloaded. Having data on the common message bus enables other clients on the project to consume the same data and not interfere with our software.

Importing image data into NumPy arrays

We are going to demonstrate how to do image processing using Python's libraries such as NumPy and SciPy.

In scientific computing, images are usually seen as n -dimensional arrays. They are usually two-dimensional arrays; in our examples, they are represented as a NumPy array data structure. Therefore, functions and operations performed on those structures are seen as matrix operations.

Images in this sense are not always two-dimensional. For medical or bio-sciences, images are data structures of higher dimensions, such as 3D (having the z axis as depth or as the time axis) or 4D (having three spatial dimensions and a temporal one as the fourth dimension). We will not be using those in this recipe.

We can import images using various techniques; they all depend on what you want to do with image. Also, it depends on the larger ecosystem of tools you are using and the platform you are running your project on.

In this recipe we will demonstrate several ways to use image processing in Python, mainly related to scientific processing and less on the artistic side of image manipulation.

Getting ready

In some examples in this recipe, we use the SciPy library, which you have already installed if you have installed NumPy. If you haven't, it is easily installable using your OS's package manager by executing the following command:

\$ sudo apt-get install python-scipy

For Windows users, we recommend using prepackaged Python environments, such as EPD, that we discussed in *Chapter 1, Preparing Your Working Environment*.

If you want to install these using official source distributions, make sure you have installed system dependencies, such as:

f BLAS and LAPACK: libblas and liblapack

f C and Fortran compilers: gcc and gfortran

How to do it...

Whoever has worked in the field of digital signal processing or even attended a university course on this or a related subject must have come across Lena's image, the de facto standard image, used for demonstrating image processing algorithms.

SciPy contains this image already packed inside the `misc` module, so it is really simple for us to re-use that image. This is how to read and show this image:

```
import scipy.misc
import matplotlib.pyplot as plt
# load already prepared ndarray from scipy lena = scipy.misc.lena()
# set the default colormap to gray plt.gray()
```

```
plt.imshow(lena) plt.colorbar() plt.show()
```

This should open a new window with a figure displaying Lena's image in gray tones and axes. The color bar shows a range of values in the figure; here it shows 0—black to 255—white.



Further, we could examine this object with the following code:

```
print lena.shape
print lena.max()
print lena.dtype
```

The output for the preceding code is as follows:

```
(512, 512)
245
dtype('int32')
```

Here we see that the image is:

- f 512 points wide and 512 points high
- f The max value in the whole array (that is, the image) is 254
- f Every point is represented as a little endian 32-bit long integer

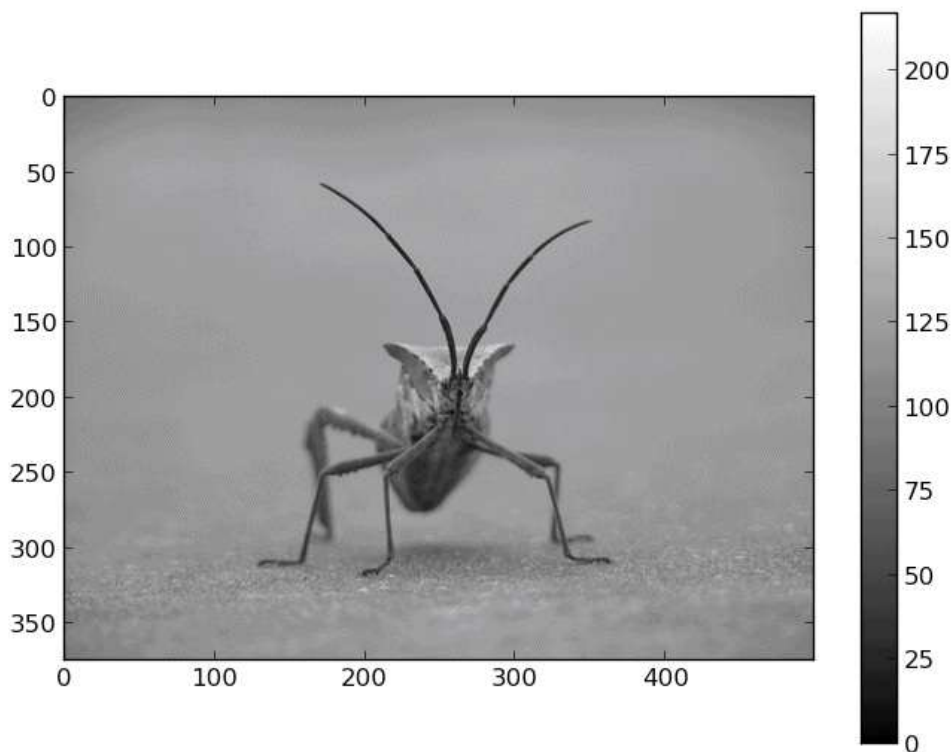
We could also read in an image using Python Imaging Library (PIL), which we installed in *Chapter 1, Preparing Your Working Environment*.

```
import numpy
import Image
import matplotlib.pyplot as plt

bug = Image.open('stinkbug.png')
arr = numpy.array(bug.getdata(), numpy.uint8).reshape(bug.size[1], bug.size[0],
3)

plt.gray()
plt.imshow(arr) plt.colorbar() plt.show()
```

We should see something similar to Lena's image as follows:



This is useful if we are already tapping into an existing system that uses PIL as their default image loader.

How it works...

Other than just loading the images, what we really want to do is use Python to manipulate images and process them. We want to be able to, say, load a real image that consists of RGB channels, convert that into one channel ndarray, and later use array slicing to also zoom in to the part of the image. Here's the code to demonstrate how we are able to use NumPy and matplotlib to do that.

```
import matplotlib.pyplot as plt
import scipy
import numpy
```

```
bug = scipy.misc.imread('stinkbug1.png')
```

```
# if you want to inspect the shape of the loaded image # uncomment following
line
```

```
#print bug.shape
```

the original image is RGB having values for all three # channels separately. We need to convert that to greyscale image # by picking up just one channel.

```
# convert to gray bug = bug[:, :, 0] bug[:, :, 0]
```

is called array slicing. This NumPy feature allows us to select any part of the multidimensional array. For example, let's see a one-dimensional array:

```
>>> a = array(5, 1, 2, 3, 4)
```

```
>>> a[2:3]
```

```
array([2])
```

```
>>> a[:2]
```

```
array([5, 1])
```

```
>>> a[3:]
```

```
array([3, 4])
```

For multidimensional arrays, we separate each dimension with a comma (.). For example:

```
>>> b = array([[1,1,1],[2,2,2],[3,3,3]]) # matrix 3 x 3
```

```
>>> b[0,:] # pick first row
```

```
array([1,1,1])
```

```
>>> b[:,0] # we pick the first column
```

```
array([1,2,3])
```

Have a look at the following code:

```
# show original image plt.figure()
```

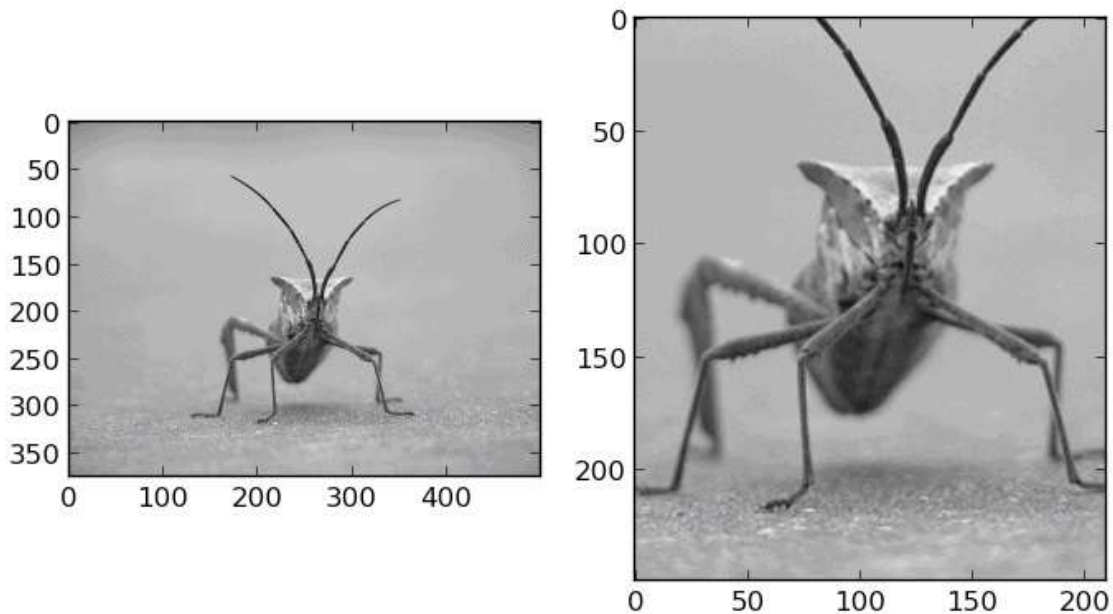
```
plt.gray()
```

```
plt.subplot(121) plt.imshow(bug)
```

```
# show 'zoomed' region zbug = bug[100:350,140:350]
```

Here we zoom into the particular portion of the whole image. Remember that the image is just a multidimensional array represented as a NumPy array. Zooming here means selecting a range of rows and columns from this matrix. So we select a partial matrix from rows 100 to 250 and columns 140 to 350. Remember that indexing starts at 0, so the row at coordinate 100 is the 101st row.

```
plt.subplot(122)
plt.imshow(zbug)
plt.show()
```

 This will be displayed as follows:

There's more...

For large images we recommend using `numpy.memmap` for memory mapping of images. This will speed up manipulating the image data. For example:

```
import numpy
file_name = 'stinkbug.png'
image = numpy.memmap(file_name, dtype=numpy.uint8, shape = (375, 500))
```

Here we load part of a large file into memory, accessing it as a NumPy array. This is very efficient and allows us to manipulate file data structures as standard NumPy arrays without loading everything into memory. The argument `shape` defines the shape of the array loaded from the `file_name` argument, which is a file-like object. Note that this is a similar concept to Python's `mmap` argument (<http://docs.python.org/2/library/mmap.html>) but is different in a very important way—NumPy's `memmap` attribute returns an array-like object, while Python's `mmap` returns a file-like object. So the way we use them is very different, yet very natural in each environment.

There are some specialized packages that just focus on image processing, for

example, scikit-image (<http://scikit-image.org/>); this is basically a free collection of algorithms for image processing, built on top of NumPy/SciPy libraries. If you want to do edge detection, remove noise from an image, or find contours, scikit is the tool to use to look for algorithms. The best way to start is to look at the example gallery and find the example image and code (http://scikit-image.org/docs/dev/auto_examples/).

Generating controlled random datasets

In this recipe, we will show different ways of generating random number sequences and word sequences. Some of the examples use standard Python modules, and some use NumPy/SciPy functions.

We will go into some statistics terminology, but we will explain every term so you don't have to have a statistical reference book with you while reading this recipe.

We generate artificial datasets using common Python modules. By doing so, we are able to understand distributions, variance, sampling, and similar statistical terminology. More importantly, we can use this fake data as a way to understand if our statistical method is capable of discovering models we want to discover. We can do that because we know the model in advance and verify our statistical method by applying it over our known data. In real life, we don't have that ability and there is always a percentage of uncertainty that we must assume, giving way to errors.

Getting ready

We don't need anything new installed on the system in order to exercise these examples. Having some knowledge of statistics is useful, although not required. To refresh our statistical knowledge, here's a little glossary we will use in this and the following chapters.

f Distribution or probability distribution: This links the outcome of a statistical experiment with the probability of occurrence of that experiment.

f Standard deviation: This is a numerical value that indicates how individuals vary in comparison to a group. If they vary more, the standard derivation will be big, and the opposite—if all the individual experiments are more or less the same across the whole group, the standard derivation will be small.

f Variance: This equals the square of standard derivation.

f Population or statistical population: This is a total set of all the potentially observable cases, for example, all the grades of all the students in the world, if we are interested in getting the student average of the world.

f Sample: This is a subset of the population. We cannot obtain all the grades of all the students in the world, so we have to gather only a sample of data and model it.

How to do it...

We can generate a simple random sample using Python's `random` module. Here's an example:

```
import pylab
```

```
import random
```

```
SAMPLE_SIZE = 100
```

```
# seed random generator
```

```
# if no argument provided
```

```
# uses system current time
```

```
random.seed()
```

```
# store generated random values here
```

```
real_rand_vars = []
```

```
# pick some random values
```

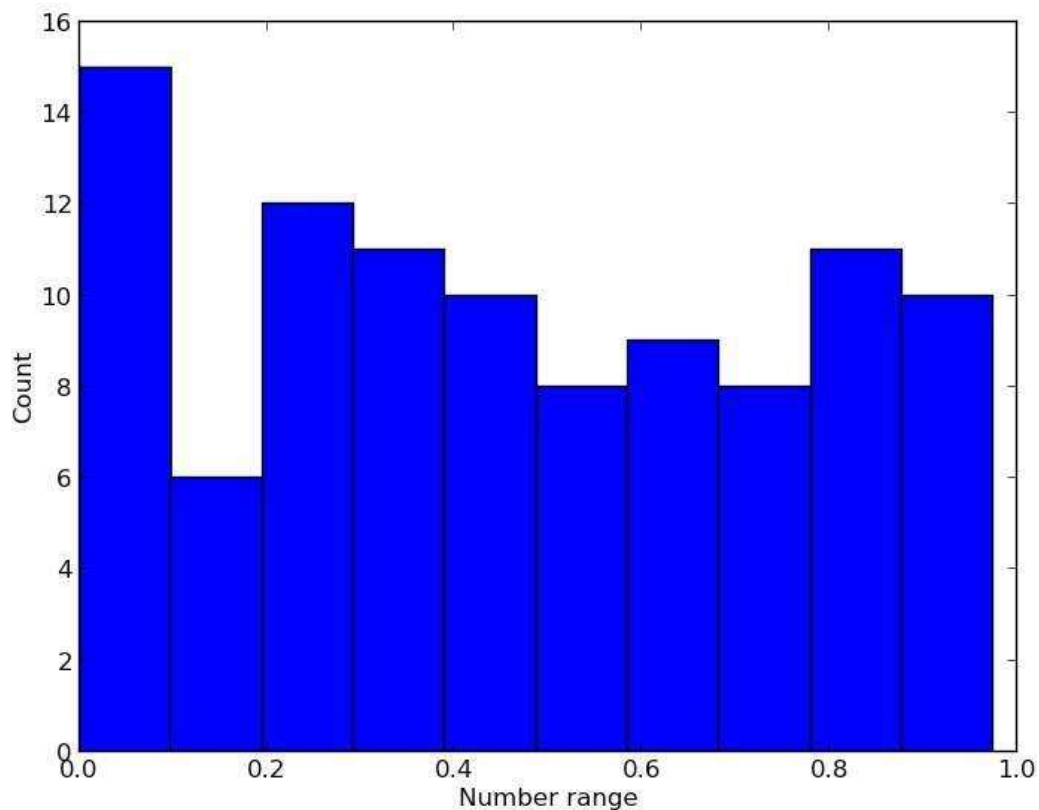
```
real_rand_vars = [random.random() for val in xrange(SAMPLE_SIZE)] # create histogram  
from data in 10 buckets
```

```
pylab.hist(real_rand_vars, 10)
```

```
# define x and y labels pylab.xlabel("Number range") pylab.ylabel("Count")
```

```
# show figure pylab.show()
```

This is a uniformly distributed sample. When we run this example, we should see something similar to the following plot:



Try setting `SAMPLE_SIZE` to a big number (say 10000) and see how the histogram behaves.

If we wanted to have values that range not from 0 to 1, but say, from 1 to 6 (simulating single dice throws, for example), we could use `random.randint(min, max)`; here, `min` and `max` are the lower and upper inclusive bounds respectively. If what you want to generate are floats and not integers, there is a `random.uniform(min, max)` function to provide that. In a similar fashion, and using the same tools, we can generate a time series plot of fictional price growth data with some random noise.

```
import pylab
import random

# days to generate data for duration = 100
# mean value
mean_inc = 0.2

# standard deviation std_dev_inc = 1.2
```

```

# time series
x = range(duration) y = []
price_today = 0

for i in x:
    next_delta = random.normalvariate(mean_inc, std_dev_inc) price_today +=
    next_delta
    y.append(price_today)

pylab.plot(x,y)
pylab.xlabel("Time") pylab.ylabel("Value") pylab.show()

```

This code defines a series of 100 data points (fictional days). For every next day, we pick a random value from the normal distribution (`random.normalvariate()`) ranging from `mean_inc` to `std_dev_inc` and add that value to yesterday's price value (`price_today`).

If we wanted more control, we could use different distributions. The following code illustrates and visualizes different distributions. We will comment separate code sections as we present them. We start by importing required modules and defining a number of histogram buckets. We also create a figure that will hold our histograms.

```

# coding: utf-8
import random
import matplotlib
import matplotlib.pyplot as plt

SAMPLE_SIZE = 1000 # histogram buckets buckets = 100

plt.figure()
# we need to update font size just for this example
matplotlib.rcParams.update({'font.size': 7})
To lay out all the required plots, we define a grid of six by two subplots for all
the histograms. The first plot is a normal distributed random variable.

plt.subplot(621)
plt.xlabel("random.random")
# Return the next random floating point number in the range [0.0, 1.0).
res = [random.random() for _ in xrange(1, SAMPLE_SIZE)]

```

```
plt.hi
```

For the second plot, we plot a uniformly distributed random variable.

```
plt.subplot(622)
plt.xlabel("random.uniform")
# Return a random floating point number N such that a <= N <= b for a <= b and
# b <= N <= a for b < a.
# The end-point value b may or may not be included in the range depending on
# floating-point rounding in the equation a + (b-a) * random().
a = 1
b = SAMPLE_SIZE
res = [random.uniform(a, b) for _ in xrange(1, SAMPLE_SIZE)] plt.hist(res,
buckets)
```

The third plot is a triangular distribution.

```
plt.subplot(623)
plt.xlabel("random.triangular")

# Return a random floating point number N such that low <= N <= high and with
# the specified # mode between those bounds. The low and high bounds default to
# zero and one. The mode
# argument defaults to the midpoint between the bounds, giving a symmetric
# distribution.
low = 1
high = SAMPLE_SIZE
res = [random.triangular(low, high) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

The fourth plot is a beta distribution. The condition on the parameters is that alpha and beta should be greater than zero. The returned values range between 0 and 1.

```
plt.subplot(624)
plt.xlabel("random.betavariate")
alpha = 1
beta = 10
res = [random.betavariate(alpha, beta) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```


The fifth plot visualizes an exponential distribution. `lamdb` is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called `lambda`, but that is a reserved word in Python.) The returned values range from 0 to positive infinity if `lamdb` is positive, and from negative infinity to 0 if `lamdb` is negative.

```
plt.subplot(625)
plt.xlabel("random.expovariate")
lamdb = 1.0 / ((SAMPLE_SIZE + 1) / 2.)
res = [random.expovariate(lamdb) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

Our next plot is the gamma distribution, where the condition on the parameters is that `alpha` and `beta` are greater than 0. The probability distribution function is:

$$\frac{x^{a-1} e^{-x}}{\Gamma(a) \beta^a}$$

PDF $x(a) = \frac{x^{a-1} e^{-x}}{\Gamma(a) \beta^a}$

β

Here's the code for the gamma distribution:

```
plt.subplot(626)
plt.xlabel("random.gammavariate")

alpha = 1
beta = 10
res = [random.gammavariate(alpha, beta) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

Log normal distribution is our next plot. If you take the natural logarithm of this distribution, you'll get a normal distribution with the mean `mu` and the standard deviation `sigma`. `mu` can have any value, and `sigma` must be greater than zero.

```
plt.subplot(627)
plt.xlabel("random.lognormvariate")
mu = 1
sigma = 0.5
res = [random.lognormvariate(mu, sigma) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

The next plot is normal distribution, where mu is the mean and sigma is the standard deviation.

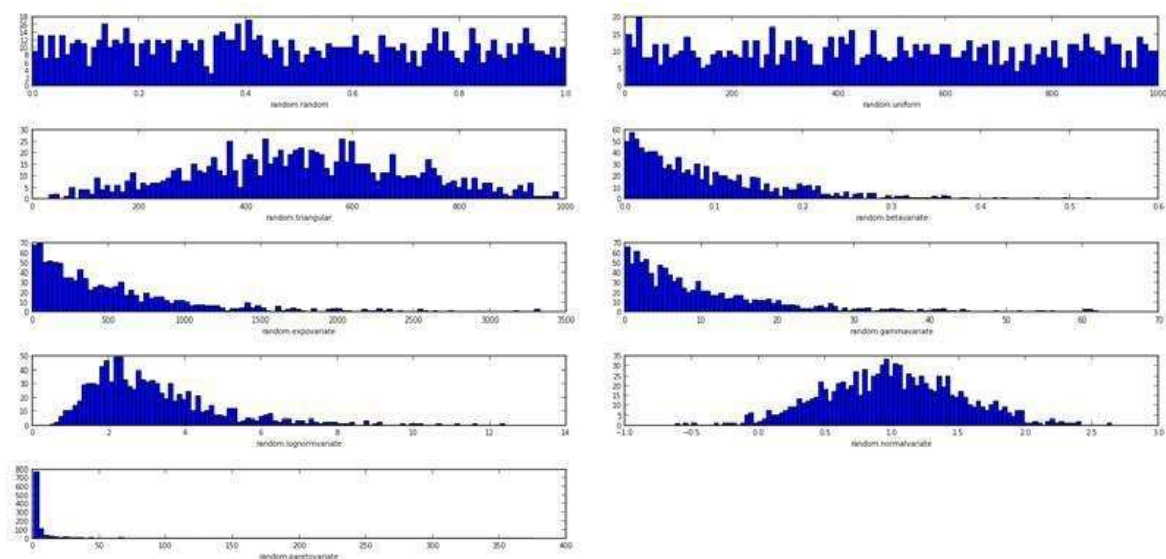
```
plt.subplot(628)
plt.xlabel("random.normalvariate")
mu = 1
sigma = 0.5
res = [random.normalvariate(mu, sigma) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

The last plot is the Pareto distribution. alpha is the shape parameter.

```
plt.subplot(629)
plt.xlabel("random.paretovariate")
alpha = 1
res = [random.paretovariate(alpha) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

```
plt.tight_layout() plt.show()
```

This was a big code example, but basically, we pick 1,000 random numbers according to various distributions. These are common distributions used in different statistical branches (economics, sociology, bio-sciences, and so on). We should see differences in the histogram based on the distribution algorithm used. Take a moment to understand the following nine plots:



Use `seed()` to initialize the pseudo-random generator, so `random()` produces the same expected random values. This is sometimes useful and it is better than pregenerating random data and saving it to a file. The latter technique is not always feasible as it requires saving (possibly huge amounts of) data on a filesystem.

If you want to prevent any repeatability of your randomly generated sequences, we recommend using `random.SystemRandom`, which uses `os.urandom` underneath; `os.urandom` provides access to more entropy sources. If using this random generator interface, `seed()` and `setstate()` have no effect; hence, these samples are not reproducible.

If we want to have some random words, the easiest way (on Linux) is probably to use `/usr/share/dict/words`. We can see how that is done, in the following example:

```
import random

with open('/usr/share/dict/words', 'rt') as f:
    words = f.readlines()
words = [w.rstrip() for w in words]

for w in random.sample(words, 5):
    print w
```

This solution is for Unix only and will not work on Windows (it will work on Mac OS, though). For Windows, you could use a file constructed from various free sources (Project Gutenberg, Wiktionary, British National Corpus, or <http://norvig.com/big.txt> by Dr Peter Norvig).

Smoothing the noise in real-world data

In this recipe, we introduce a few advanced algorithms to help with cleaning the data coming from real-world sources. These algorithms are well known in the signal processing world, and we will not go deep into mathematics but will just exemplify how and why they work and for what purposes they can be used.

Getting ready

Data that comes from different real-life sensors usually is not smooth and clean

and contains some noise that we usually don't want to show on diagrams and plots. We want graphs and plots to be clear and to display information and cost viewers minimal efforts to interpret.

We don't need any new software installed because we are going to use some already familiar Python packages: NumPy, SciPy, and matplotlib.

How to do it...

The basic algorithm is based on using the rolling window (for example, convolution). This window rolls over the data and is used to compute the average over that window.

For our discrete data, we use NumPy's convolve function; it returns a discrete linear convolution of two one-dimensional sequences. We also use NumPy's linspace function, which generates a sequence of evenly spaced numbers for a specified interval.

The function ones defines an array or matrix (for example, a multidimensional array) where every element has the value 1. This helps with generating windows for use in averaging.

How it works...

One simple and naive technique to smooth the noise in data we are processing is to average over some window (sample) and plot just that average value for the given window, instead of all the data points. This is the basis for more advanced algorithms.

```
from pylab import *  
from numpy import *
```

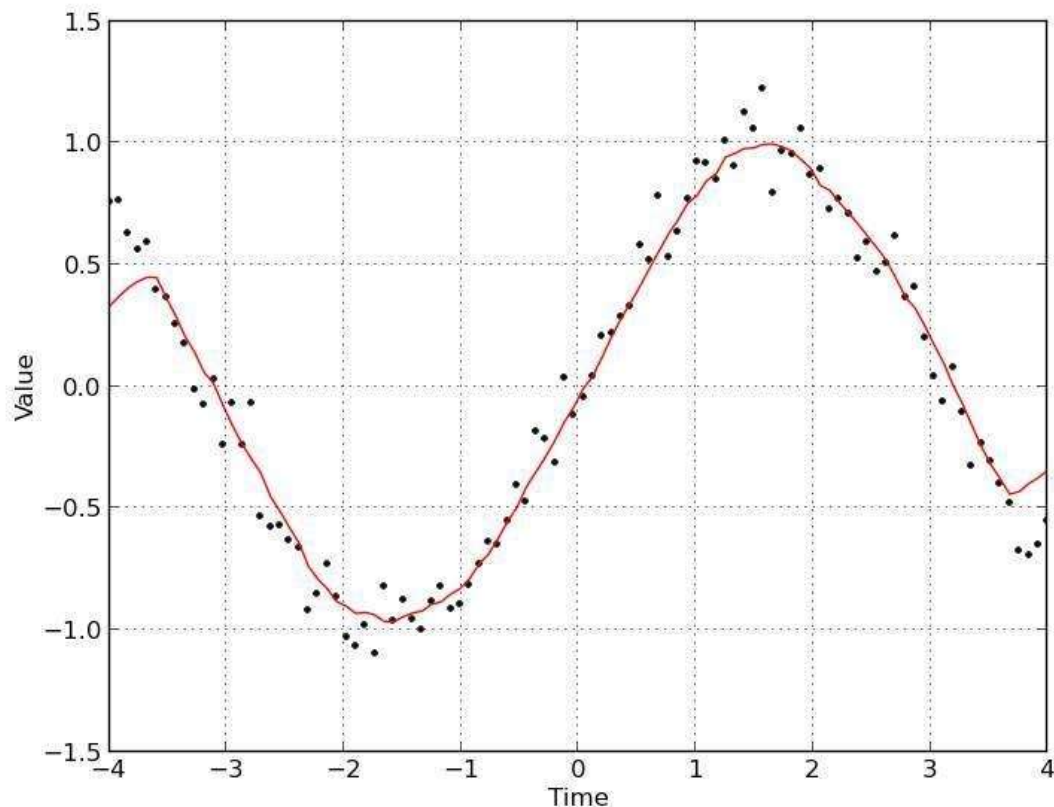
```
def moving_average(interval, window_size): '''Compute convoluted window for  
given size '''  
window = ones(int(window_size)) / float(window_size) return  
convolve(interval, window, 'same')
```

```
t = linspace(-4, 4, 100) y = sin(t) + randn(len(t))*0.1  
plot(t, y, "k.")
```

```
# compute moving average y_av = moving_average(y, 10) plot(t, y_av, "r")  
#xlim(0,1000)
```

```
xlabel("Time") ylabel("Value") grid(True)
show()
```

Here, we show how the smoothed line looks compared to the original data points (plotted as dots):



Following on this idea, we can jump ahead to an even more advanced example and use the existing SciPy library to make this window smoothing work even better.

The method we are going to demonstrate is based on convolution (summation of functions) of a scaled window with the signal (that is, data points). This signal is prepared in a clever way, adding copies of the same signal on both ends but reflecting it, so we minimize the boundary effect. This code is based on SciPy Cookbook's example that can be found here:
<http://www.scipy.org/Cookbook/SignalSmooth>.

```
import numpy
```

```

from numpy import *
from pylab import *

# possible window type
WINDOWS = ['flat', 'hanning', 'hamming', 'bartlett', 'blackman'] # if you want to
see just two window type, comment previous line, # and uncomment the
following one
# WINDOWS = ['flat', 'hanning']

def smooth(x, window_len=11, window='hanning'): """
Smooth the data using a window with requested size. Returns smoothed signal.

x -- input signal
window_len -- lenght of smoothing window
window -- type of window: 'flat', 'hanning', 'hamming',
'bartlett', 'blackman'
flat window will produce a moving average smoothing. """
if x.ndim != 1:
    raise ValueError, "smooth only accepts 1 dimension arrays."
if x.size < window_len:
    raise ValueError, "Input vector needs to be bigger than window size."
if window_len < 3: return x
if not window in WINDOWS:
    raise ValueError("Window is one of 'flat', 'hanning',
'hamming', "
'bartlett', 'blackman'") # adding reflected windows in front and at the end
s=numpy.r_[x[window_len-1:0:-1], x, x[-1:-window_len:-1]] # pick windows
type and do averaging
if window == 'flat': #moving average

w = numpy.ones(window_len, 'd')
else:
# call appropriate function in numpy
w = eval('numpy.' + window + '(window_len)')

# NOTE: length(output) != length(input), to correct this: # return
y[(window_len/2-1):-(window_len/2)] instead of just y. y =
numpy.convolve(w/w.sum(), s, mode='valid')

```

```

return y

# Get some evenly spaced numbers over a specified interval. t = linspace(-4, 4,
100)

# Make some noisy sinusoidal x = sin(t)
xn = x + randn(len(t))*0.1

# Smooth it y = smooth(x)
# windows ws = 31
subplot(211) plot(ones(ws))
# draw on the same axes hold(True)
# plot for every windows for w in WINDOWS[1:]:
eval('plot('+w+'(ws) )') # configure axis properties axis([0, 30, 0, 1.1])
# add legend for every window legend(WINDOWS)
title("Smoothing windows")

# add second plot subplot(212)
# draw original signal plot(x)

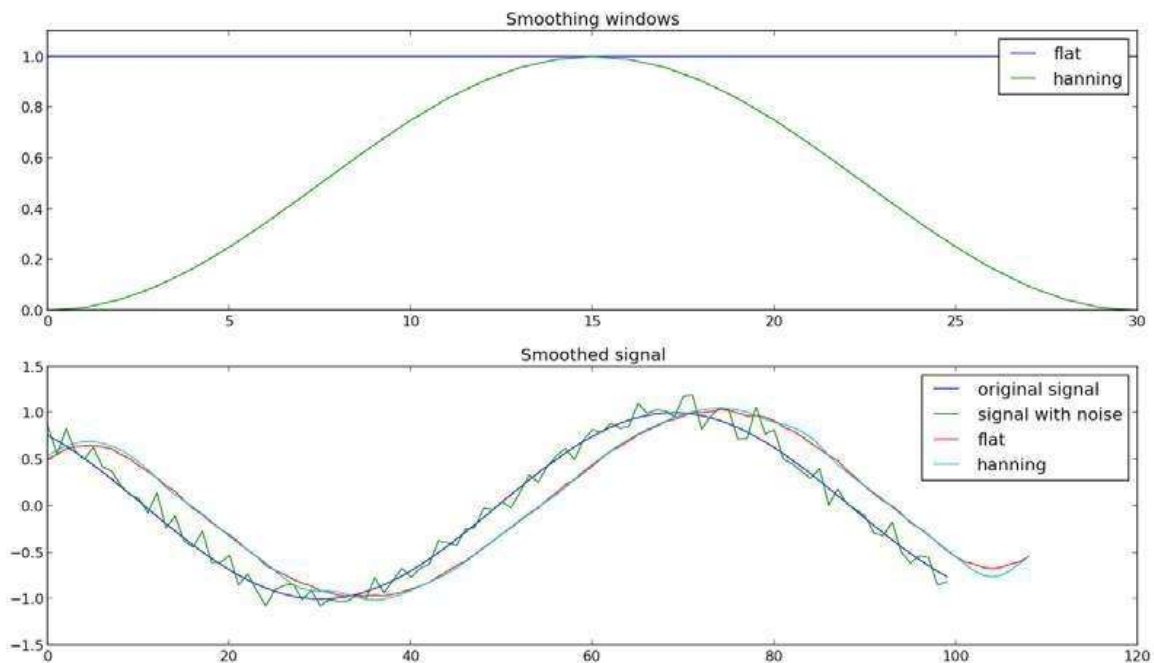
# and signal with added noise plot(xn)
# smooth signal with noise for every possible windowing algorithm for w in
WINDOWS:
plot(smooth(xn, 10, w))

# add legend for every graph
l=['original signal', 'signal with noise'] l.extend(WINDOWS)
legend(l)

title("Smoothed signal")
show()

```

We should see the following two plots to see how the windowing algorithm influences the noise signal. The top plot represents possible windowing algorithms and the bottom one displays every possible result, from the original signal, to the noised up signal, and even the smoothed signal for every windowing algorithm. Try commenting possible window types and leave just one or two to gain better understanding.



There's more...

Another very popular signal smoothing algorithm is Median Filter. The main idea of this filter is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. This idea makes this filter fast and usable both for one-dimensional datasets as well as for two-dimensional datasets (such as images).

In the following example, we use the implementation from the SciPy signal toolbox:

```
import numpy as np
import pylab as p
import scipy.signal as signal

# get some linear data
x = np.linspace(0, 1, 101)
# add some noisy signal
x[3::10] = 1.5

p.plot(x)
p.plot(signal.medfilt(x,3))
```

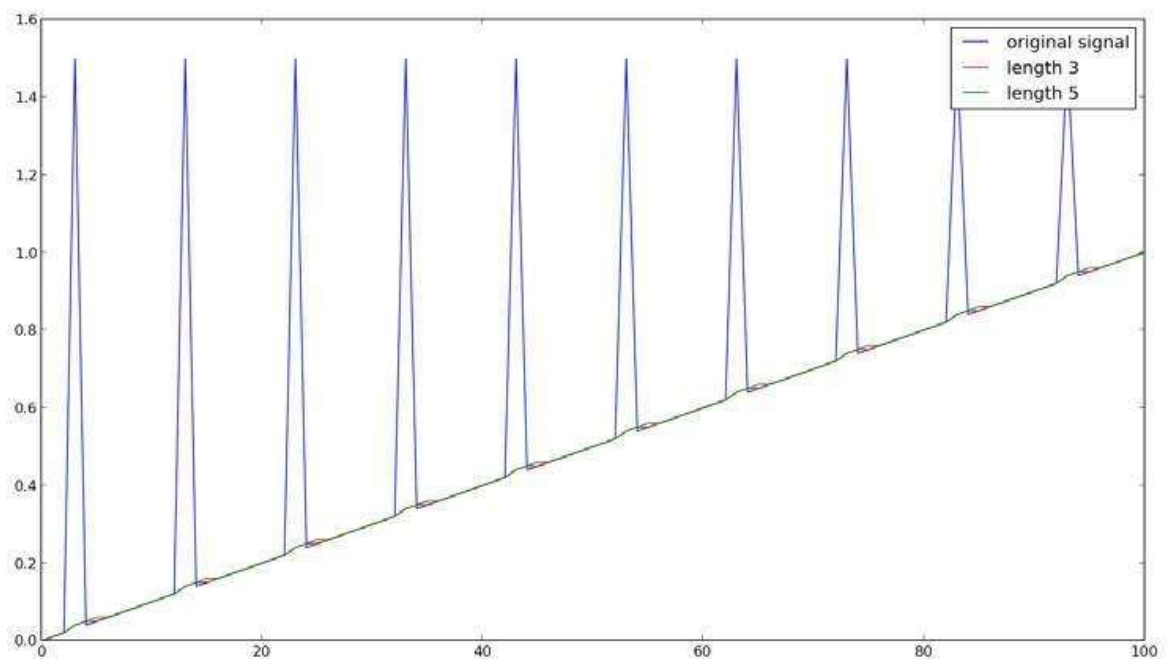


```
p.plot(signal.medfilt(x,5))
```

```
p.legend(['original signal', 'length 3','length 5'])
```

```
p.show ()
```

We see in the following plot that the bigger the window, the more our signal gets distorted as compared to the original but the smoother it looks:



There are many more ways to smooth data (signals) that you receive from external sources. It depends a lot on the area you are working in and the nature of the signal. Many algorithms are specialized for a particular signal, and there may not be a general solution for every case you encounter.

There is, however, one important question: "When should you not smooth a signal?" One common situation where you should not smooth signals is prior to statistical procedures, such as least-squares curve fitting, because all smoothing algorithms are at least slightly lossy and they change the signal shape. Also, smoothed noise may be mistaken for an actual signal.

3

Drawing Your First Plots and Customizing Them

In this chapter we will go into a lot more detail and present most of the possibilities of matplotlib. We will cover:

- f Defining plot types – bar, line, and stacked charts
- f Drawing simple sine and cosine plots
- f Defining axis lengths and limits
- f Defining plot line styles, properties, and format strings
- f Setting ticks, labels, and grids
- f Adding legends and annotations
- f Moving spines to the center
- f Making histograms
- f Making bar charts with error bars
- f Making pie charts count
- f Plotting with filled areas
- f Drawing scatter plots with colored markers

Introduction

Although we have already drawn our first plots using matplotlib, we didn't go into the details about how they work, how to set them up, or what the possibilities are with using matplotlib. We explore and exercise most common types of data visualizations: line graphs, bar charts, histograms, pies, and variations thereof.

matplotlib is a powerful toolbox that satisfies almost all our needs for 2D and some 3D plotting needs as well. The best way the authors intend for you to learn matplotlib is through examples. When we need to draw a plot, we look for a similar example and try to change it to fit our needs. In this way, we are also going to present you with some useful examples and believe that this example will help you find a plot most similar to what you need.

Defining plot types – bar, line, and stacked charts

In this recipe, we will present different basic plots and what are they used for. Most of the plots described here are used daily, and some of them present the basis for understanding more advanced concepts in data visualization.

Getting ready

We start with some common charts from the `matplotlib.pyplot` library with just sample datasets to get started with basic charting and lay down the foundations of the following recipes.

How to do it...

We start by creating a simple plot in IPython. IPython is great because it allows us to interactively change plots and see the results immediately.

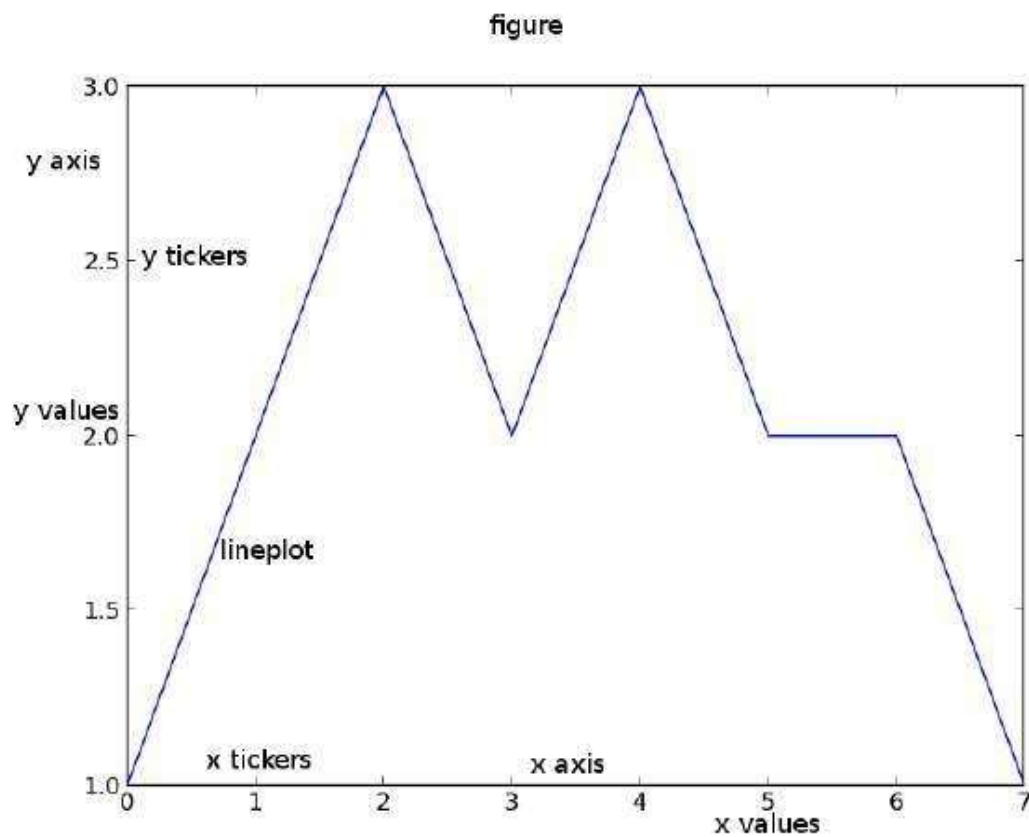
1. Start IPython by typing the following at the command prompt: **\$ ipython --pylab**

2. Then type the matplotlib plot code:

```
In [1]: plot([1,2,3,2,3,2,2,1])
```

```
Out[1]: [<matplotlib.lines.Line2D at 0x412fb50>]
```

The plot should open in a new window displaying the default look of the plot and some supporting information:



The basic plot in matplotlib contains the following elements:

f x and y axes: These are both horizontal and vertical axes.

f x and y tickers: These are little tickers denoting the segments of axes. There can be major and minor tickers.

f x and y tick labels: These represent values on particular axis.

f Plotting area: This is where the actual plots are drawn.

You will notice that the values we provided to plot() are y axis values.

plot() provides default values for the x axis; they are linear values from 0 to 7 (the number of y values minus 1).

Now, try adding values for the x axis; as first argument to the plot() function, again in the same IPython session, type:

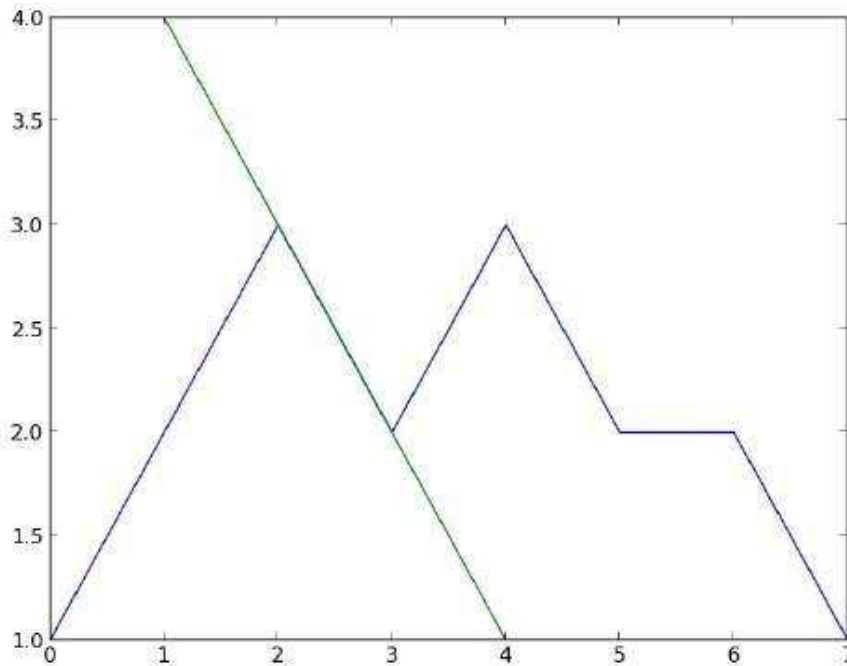
```
In [2]: plot([4,3,2,1],[1,2,3,4])
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x31444d0>]
```

Note how IPython counts input and output lines (In [2] and Out [2]). This will help us remember where we are in the current session, and enables more advanced features such as saving part of the session in a Python file. During data

analysis, using IPython for prototyping is the fastest way to come to a satisfying solution and then save particular sessions into a file, to be executed later if you need to reproduce the same plot.

This will update the plot to look like this:



We see here how matplotlib expands the y axis to accommodate the new value range and automatically changes color of the second plot line to enable us to distinguish the new plot.

Unless we turn off the hold property (by calling `hold(False)`) all subsequent plots will draw over the same axes. This is the default behavior in pylab mode in IPython, while in regular Python scripts, hold is off by default.

Let us pack some more common plots and compare them over the same dataset. You can type this in IPython or run it from a separate Python script:

```
from matplotlib.pyplot import *
```

```
# some simple data
x = [1,2,3,4]
y = [5,4,3,2]
# create new figure figure()

# divide subplots into 2 x 3 grid # and select #1
```

```
subplot(231)
plot(x, y)

# select #2 subplot(232) bar(x, y)

# horizontal bar-charts subplot(233)
barh(x, y)

# create stacked bar charts subplot(234)
bar(x, y)

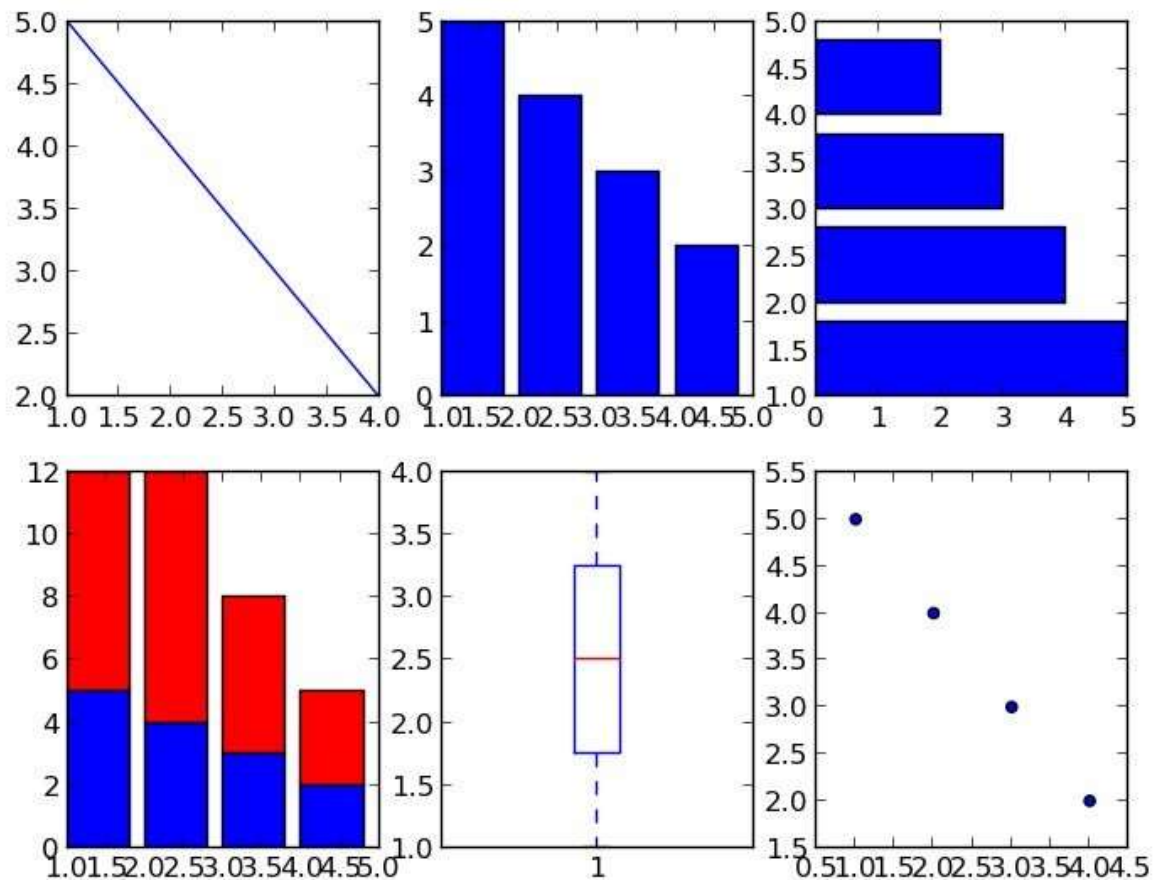
# we need more data for stacked bar charts y1 = [7,8,5,3]
bar(x, y1, bottom=y, color = 'r')

# box plot subplot(235) boxplot(x)

# scatter plot subplot(236) scatter(x,y)

show()
```

This is how it should turn out into graphs:



How it works...

With `figure()` we create a new figure. If we supply a string argument, such as `sample charts`, it will be the backend title of a window. If we call the `figure()` function with the same parameter (that can also be a number), we will make the corresponding figure active and all the following plotting will be performed on that figure.

Next, we divide the figure into a 2 by 3 grid using a `subplot(231)` call. We could call this using `subplot(3, 2, 1)`, where the first parameter is the number of rows, the second is the number of columns, and the third represents the plot number.

We continue and create a common charting type using simple calls to create vertical bar charts (`bar()`) and horizontal bars (`barh()`). For stacked bar charts, we need to tie two bar chart calls together. We do that by connecting the second bar chart with the previous using the parameter `bottom = y`.

Box plots are created using the `boxplot()` call, where the box extends from lower

to upper quartiles with the line at the median value. We will return to box plots shortly.

We finally create a scatter plot to give you an idea of a point-based dataset. This is probably more appropriately used when we have thousands of data points in a dataset, but here we wanted to illustrate the difference in representations of the same dataset.

There's more...

We can return to box plots now as we need to explain the most important display options.

For starters, we could add whiskers that extend from the box to represent the whole range of the dataset. Box and whiskers plots are mainly used to represent variations of data in one or multiple datasets; they are easy to compare and easy to read. In the same plot they can represent five statistics:

- f Minimum value: This is in the dataset

- f Second quartile: Below this the lower 25 percent of the given dataset lies f

Median value: This is the median value of the dataset

- f Third quartile: Above this the upper 25 percent of the given dataset lies f

Maximum value: This is the maximum value of the given dataset

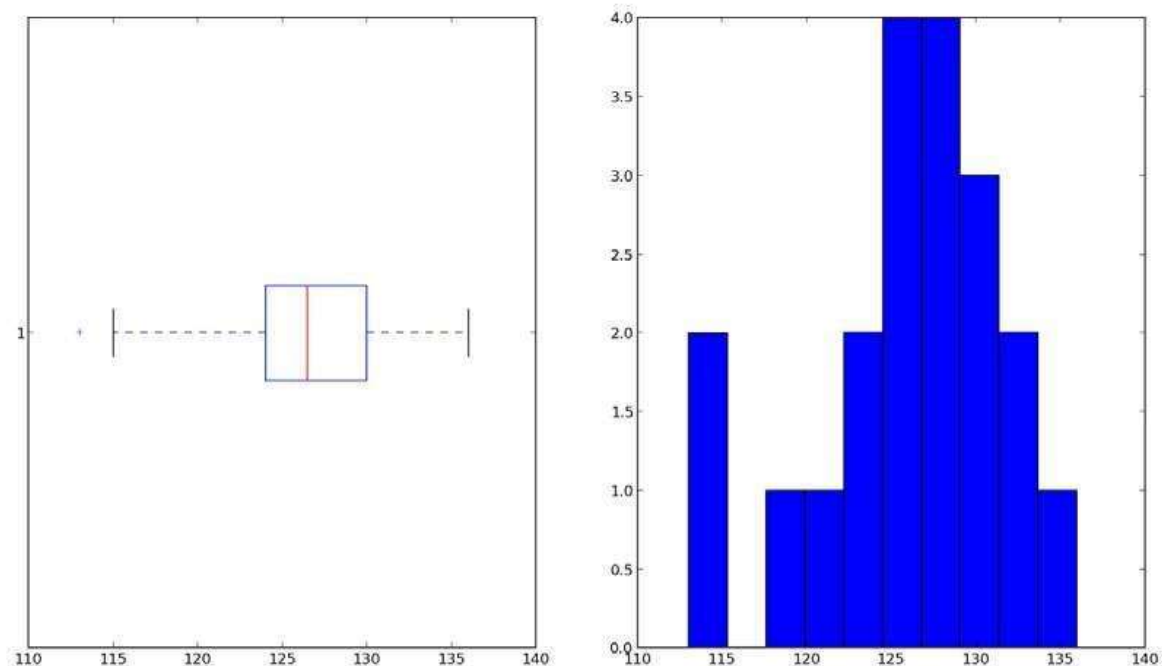
To illustrate this behavior, we will demonstrate plotting the same dataset in a box plot and a histogram as in the following code:

```
from pylab import *
```

```
dataset = [113, 115, 119, 121, 124,  
124, 125, 126, 126, 126,  
127, 127, 128, 129, 130,  
130, 131, 132, 133, 136]
```

```
subplot(121)  
boxplot(dataset, vert=False)  
subplot(122) hist(dataset)  
show()
```

That will give us the following plots:



In the preceding comparison, we can observe a difference in representation of the same dataset in two different charts. The one on the left points toward the five mentioned statistical values, while the one on the right (the histogram) displays the grouping of the dataset in a given range.

Drawing a simple sine and cosine plot

This recipe will go over basics of plotting mathematical functions and several things that are related to math graphs, such as writing Greek symbols in labels and on curves.

Getting ready

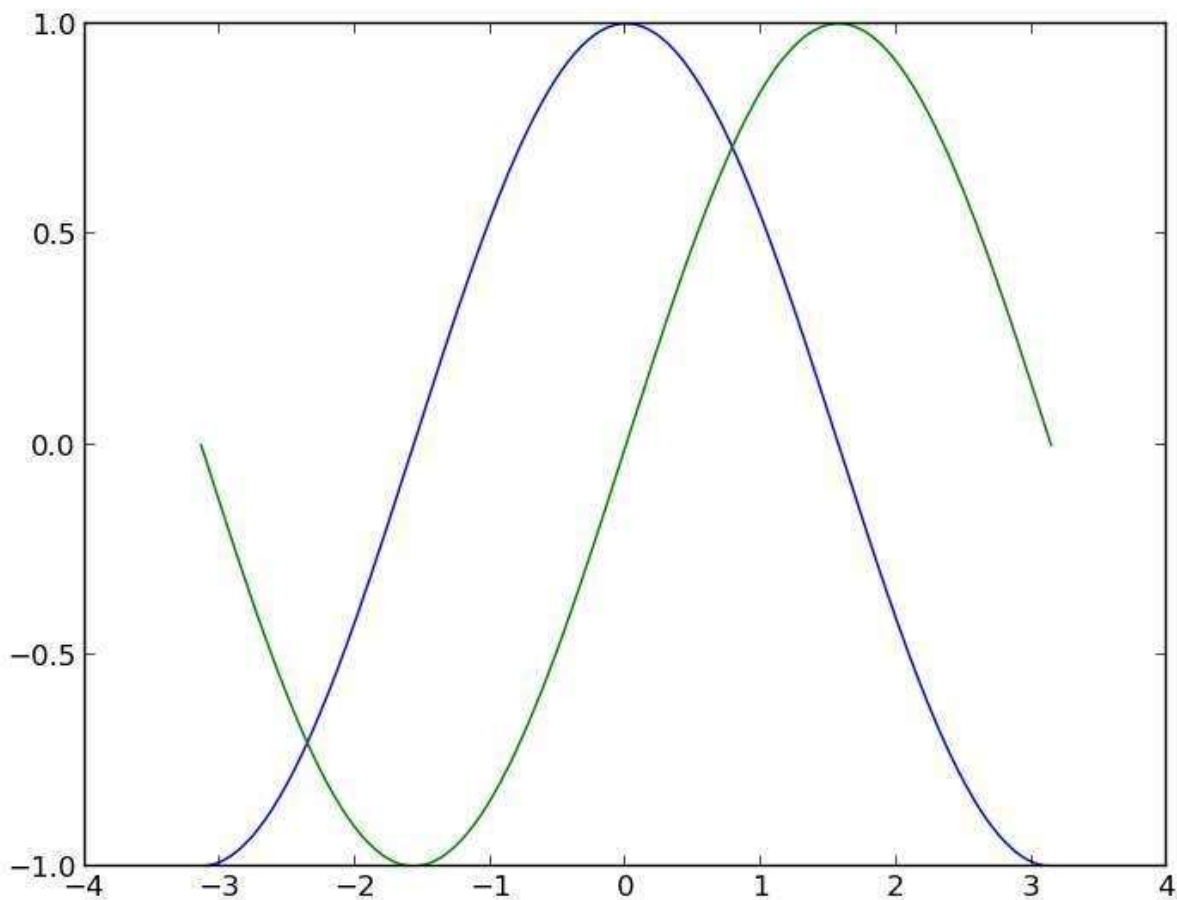
The most common graph we will use is the line plot command, which draws the given (x,y) coordinates on a figure plot.

How to do it...

We start with computing sine and cosine functions over the same linear interval—from π to π , with 256 points in between—and we plot the values for $\sin(x)$ and $\cos(x)$ over the same plot:

```
import matplotlib.pyplot as pl
import numpy as np
x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
y = np.cos(x) y1 = np.sin(x)
pl.plot(x,y) pl.plot(x, y1)
```

plt.show() That will give us the following graph:



Following this simple plot, we can customize more to give more information and be more precise about axes and boundaries:

```
from pylab import *  
import numpy as np
```

```
# generate uniformly distributed  
# 256 points from -pi to pi, inclusive  
x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
```

```
# these are vectorised versions  
# of math.cos, and math.sin in built-in Python maths # compute cos for every x  
y = np.cos(x)
```

```
# compute sin for every x y1 = np.sin(x)  
# plot cos plot(x, y)  
# plot sin plot(x, y1)
```

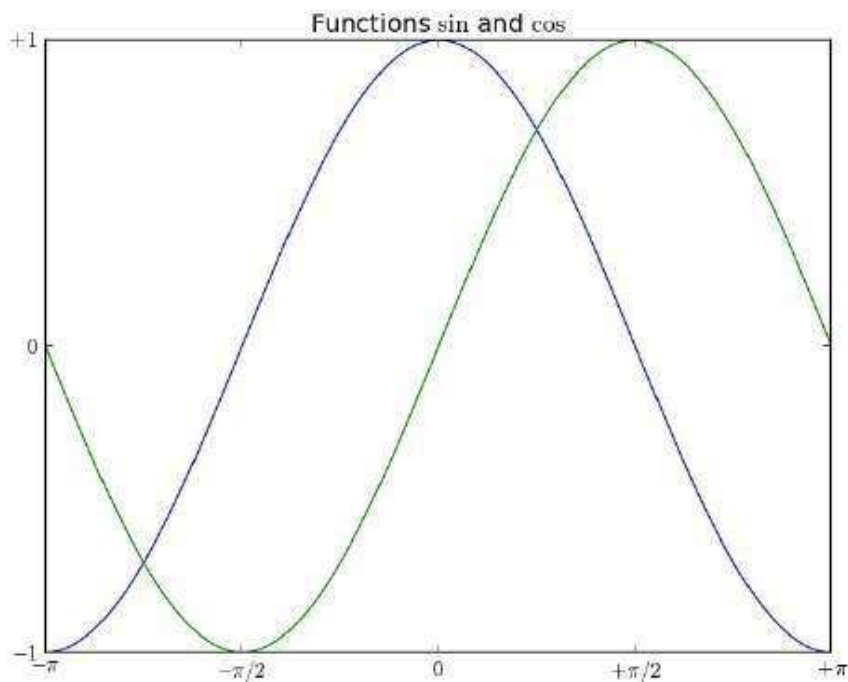
```
# define plot title
title("Functions  $\sin$  and  $\cos$ ")

# set x limit xlim(-3.0, 3.0) # set y limit ylim(-1.0, 1.0)

# format ticks at specific values
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
yticks([-1, 0, +1],
[r'$-1$', r'$0$', r'$+1$'])

show()
```

That should give us a slightly nicer graph:



We see that we used expressions such as \sin , or $-\pi$ to write letters of the Greek alphabet in figures. This is LaTeX syntax, which we will explore further in the following chapters. Here, we just illustrated how easy it is to make your math charts more readable for certain audiences.

Defining axis lengths and limits

This recipe will demonstrate a variety of useful axis properties around limits and lengths that we can configure in matplotlib.

Getting ready

For this recipe we want to fire up IPython:

```
$ ipython --pylab
```

How to do it...

Start experimenting with various properties of axes. Just calling an empty `axis()` function will return default values for the axis:

```
In [1]: axis()
```

```
Out[1]: (0.0, 1.0, 0.0, 1.0)
```

Note that if you are in interactive mode and are using a windowing backend, a figure with an empty axis will be displayed.

Here the values represent `xmin`, `xmax`, `ymin`, and `ymax` respectively. Similarly, we can set values for the `x` and `y` axes:

```
In [2]: l = [-1, 1, -10, 10]
```

```
In [3]: axis(l)
```

```
Out[3]: [-1, 1, -10, 10]
```

Again, if you are in interactive mode, this will update the same figure.

Furthermore, we can also update any value separately using keyword arguments (`**kwargs`), setting just `xmax` to a certain value.

How it works...

If we don't use `axis()` or other settings, matplotlib will automatically use minimum values that allow us to see all data points on one plot. If we set `axis()` limits to be less than the maximum values in a dataset, matplotlib will do as told and we will not see all points on the figure. This can be a source of confusion or even error, where we think we see everything we drew. One way to avoid this is to call `autoscale()` (`matplotlib.pyplot.autoscale()`), which will compute the optimal size of the axes to fit the data to be displayed.

If we want to add new axes to the same figure, we can use `matplotlib.pyplot.axes()`. We usually want to add some properties to this default call; for example, `rect`—which can have the attributes `left`, `bottom`, `width`, and `height` in normalized units (0, 1)—and maybe `axisbg`, which specifies the background color of axes.

There are also other properties that we can set for added axes such as

sharex/sharey, which accepts values for other instances of axes and share the current axis (x/y) with other axes. Or parameter polar that defines whether we want to use polar axes.

Adding new axes can be useful; for example, to combine multiple charts on one figure if there is a need to tightly couple different views on the same data to illustrate its properties. If we want to add just one line to the current figure, we can use `matplotlib.pyplot.axhline()` or `matplotlib.pyplot.axvline()`. The functions `axhline()` and `axvline()` will draw horizontal and vertical lines across axes for given x and y data values respectively. They share similar parameters, the most important ones being y position, xmin, and xmax for `axhline()` and x position, ymin, and ymax for `axvline()`.

Let's see how it looks as a figure, continuing in the same IPython session: **In [3]:**

`axhline()`

Out[3]: <matplotlib.lines.Line2D at 0x414ecd0>

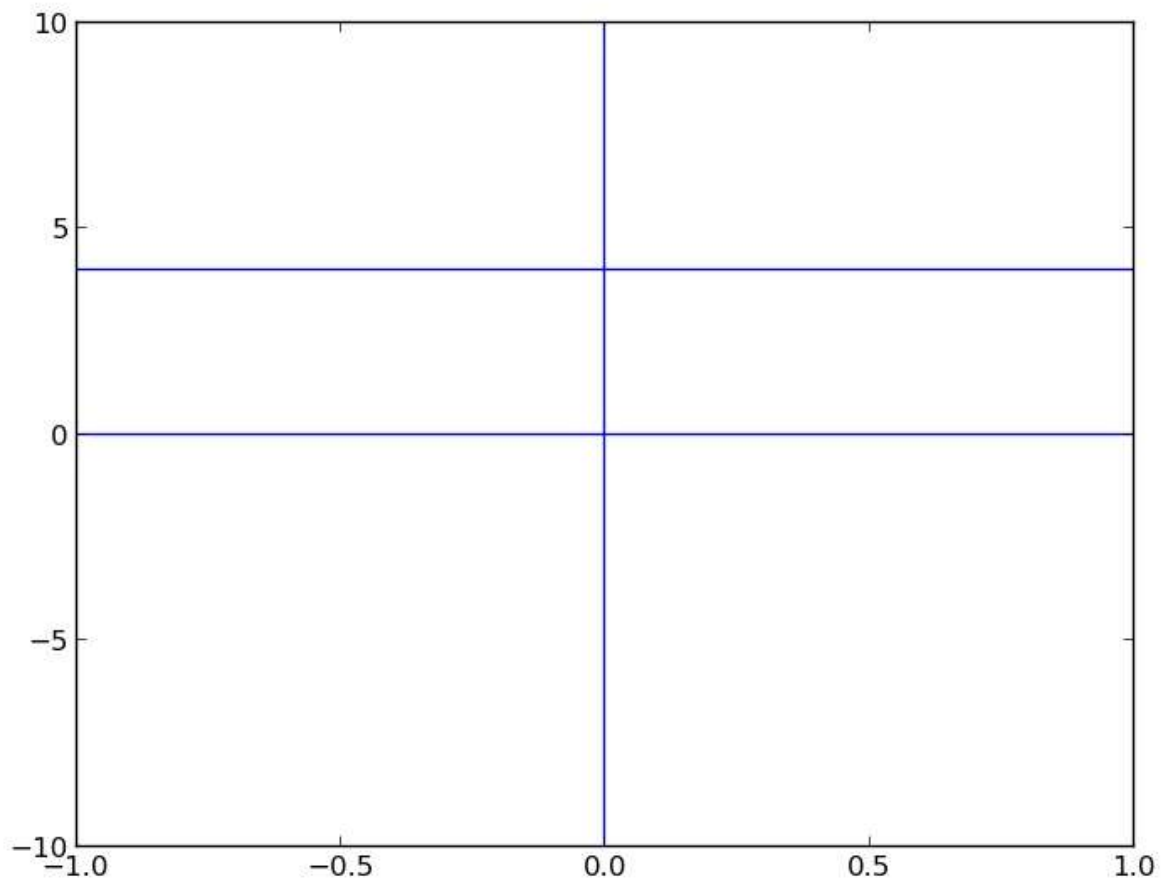
In [4]: **`axvline()`**

Out[4]: <matplotlib.lines.Line2D at 0x4152490>

In [5]: **`axhline(4)`**

Out[5]: <matplotlib.lines.Line2D at 0x4152850>

We should have a figure like the following plot:



Here we see that just calling these functions without parameters makes them take default values and draw a horizontal line for $y=0$ (`axhline()`) and a vertical line for $x=0$ (`axvline()`).

Similar to these are two related functions that allow us to add a horizontal span (rectangle) across the axes. These are `matplotlib.pyplot.axhspan()` and `matplotlib.pyplot.axspan()`. The function `axhspan()` has `ymin` and `ymax` as required parameters that define how wide the horizontal span is. Analogous to this, `axvspan()` has `xmin` and `xmax` to define the width of the vertical span.

There's more...

Having a grid in a figure is turned off by default, but it can easily be switched on and customized. A default call to `matplotlib.pyplot.grid()` will toggle the grid's visibility. Other parameters for control are as follows:

`fig` which: This defines what grid tick type to draw (can be major, minor, or both) `f`

axis: This defines which set of grid lines are drawn (can be both, x, or y)

Axes are usually controlled via `matplotlib.pyplot.axis()`. Internally, axes are represented by several Python classes, the parent one is `matplotlib.axes.Axes`, which contains most methods to manipulate axes. A single axis is represented by the `matplotlib.axis.Axis` class, where the x axis uses `matplotlib.axis.XAxis` and the y axis uses the `matplotlib.axis.YAxis` class.

We don't need to use these to perform our recipe, but it is important to know where to look if more advanced axis control interests us and when we hit the limits of what is available via the `matplotlib.pyplot` namespace.

Defining plot line styles, properties, and format strings

This recipe shows how we can change various line properties, such as styles, colors, or width. Having lines set up appropriately—according to the information presented—and distinct enough for target audiences (if the audience is younger population, we may want to target them with more vivid colors; if they are older we may want to use more contrasting colors) can make the difference between being barely noticeable and leaving a great impact on the viewer.

Getting ready

Although we stressed how important is to aesthetically tune your presentation, we first must learn how to do it.

If you don't have a particular eye for color matching, there are free and commercial online tools that can generate color sets for you. One of the most well known is Colorbrewer2, which can be found at <http://colorbrewer2.org/>.

Some serious research has been conducted on the usage of color in data visualizations, but explaining that theory is out of the scope of this book. The material on the topic is a must read if you are working with more advanced visualizations daily.

How to do it...

Let us learn how to change line properties. We can change the lines in our plots using different methods and approaches.

First and most common is to define lines by passing keyword parameters to functions such as `plot()`:

```
plot(x, y, linewidth=1.5)
```

Because a call to `plot()` returns the line instance (`matplotlib.lines.Line2D`), we can use a set of setter methods on that instance to set various properties:

```
plot(x, y)
line.set_linewidth(1.5)
```

Those who used MATLAB® will feel the need to use a third way of configuring line properties—using the `setp()` function:

```
lines = plot(x, y)
setp(lines, 'linewidth', 1.5)
```

Another way to use `setp()` is:

```
setp(lines, linewidth=1.5)
```

Whatever way you prefer to configure lines, choose one method and stay consistent for the whole project (or at least a file). This way, when you (or someone else that's you in the future) come back to the code, it will be easier to make sense of it and change it.

How it works...

All the properties we can change for a line are contained in the `matplotlib.lines.Line2D` class. We list some of them in the following table:

Property `alpha`

Value type `float`

color or `c` dashes

Any matplotlib color Sequence of on/off ink in points

label

linestyle or `ls`

linewidth or `lw` marker

`markeredgecolor` or `mec`

`markeredgewidth` or `mew`

`markerfacecolor` or `mfc`

`markersize` or `ms` `solid_capstyle`

`solid_joinstyle`

`visible` `xdata`

Any string

[`'-'` | `'--'` | `'-.'` | `':'` | `'steps'` | ...]

float value in points

Description

Sets the alpha value used for blending; not supported on all backends.

Sets the color of the line.

Sets the dash sequence, the sequence of dashes with on/off ink in points. If seq is empty or if seq = (None, None), linestyle will be set to solid.

Sets the label to s for auto legend. Sets the linestyle of the line (also accepts drawstyles).

Sets the line width in points.

[7 | 4 | 5 | 6 | 'o' Sets the line marker. | 'D' | 'h' | 'H' |

'_' | " | 'None' | '

' | None | '8' | 'p'

| ',' | '+' | '.' |

's' | '*' | 'd' | 3

| 0 | 1 | 2 | '1' |

'3' | '4' | '2' | 'v'

| '<' | '>' | '^' |

'|' | 'x' | '\$...\$' |

tuple | Nx2 array]

Any matplotlib color

Sets the marker edge color.

float value in points Sets the marker edge width in points.

Any matplotlib color Set the marker face color.

float

['butt' | 'round' | 'projecting']

['miter' | 'round' | 'bevel']

[True | False]

np.array

Set the marker size in points. Set the cap style for solid line styles.

Set the join style for solid line styles.

Set the artist's visibility.

Set the data np.array for x. Property Value type ydata np.array Zorder Any number

Description

Set the data np.array for y.

Set the z axis order for the artist.

Artists with lower Zorder values are drawn first.

If x and y are axes going horizontal to the right and vertical to the top of the screen, the z axis is the one extending toward the viewer. So 0 value would be at the screen, 1, one layer above, and so on.

The following table shows some linestyles:

Linestyle

'_'
'--'
'-.'
'.'
'.'
'None', ' ', "

Description Solid

Dashed

Dash_dot

Dotted

Draw nothing

The following table shows line markers:

Marker 'o'

'D'
'h'
'H'
' '
'_
' ', 'None', ' ', None
'8'
'p'
' '
'
'+'

'.'
's'
'*'

Description Circle

Diamond

Hexagon1

Hexagon2

Horizontal line Nothing

Octagon Pentagon Pixel

Plus

Point

Square

Star

Marker Description 'd' Thin_diamond 'v' Triangle_down '<' Triangle_left '>'

Triangle_right '^' Triangle_up '|' Vertical line 'x' X

Color

We can get all colors that matplotlib supports by calling
matplotlib.pyplot.colors(); this will give:

Alias Color b Blue g Green r Red c Cyan m Magenta y Yellow k Black w White

These colors can be used in different matplotlib functions that take color arguments. If these basic colors are not enough—and as we progress, they will not be enough—we can use two other ways of defining a color value. We can use an HTML hexadecimal string: color = '#eeeeff'

We can also use legal HTML color names ('red', 'chartreuse'). We can also pass an RGB tuple normalized to [0, 1]:

color = (0.3, 0.3, 0.4)

Argument color is accepted by a range of functions, such as title():

title('Title in a custom color', color='#123456')

Background color

By providing axisbg to a function such as matplotlib.pyplot.axes() or matplotlib.pyplot.subplot(), we can define the background color of an axis:

subplot(111, axisbg=(0.1843, 0.3098, 0.3098))

Setting ticks, labels, and grids

In this recipe we will continue with setting axis and line properties and adding more data to our figure and charts.

Getting ready

Let us learn a little about figures and subplots.

In matplotlib, `figure()` is used to explicitly create a figure, which represents a user interface window. Figures are created implicitly just by calling `plot()` or similar functions. This is fine for simple charts, but having the ability to explicitly create a figure and get a reference to its instance is very useful for more advanced use.

A figure contains one or more subplots. Subplots allow us to arrange plots in a regular grid. We already used `subplot()`, in which we specify the number of rows and columns and the number of the plot we are referring to.

If we want more control, we need to use axes instances from the `matplotlib.axes.Axes` class. They allow us to place plots at any location in the figure. An example of this would be to put a smaller plot inside a bigger one.

How to do it...

Ticks are part of figures. They consist of tick locators—where ticks appear—and tick formatters which show how ticks appear. There are major and minor ticks. Minor ticks are not visible by default. More importantly, major and minor ticks can be formatted and located independently of each other.

We can use `matplotlib.pyplot.locator_params()` to control the behavior of tick locators. Even though tick locations are usually determined automatically, we can control the number of ticks and use a tight view if we want to, for when plots are smaller.

```
from pylab import *
```

```
# get current axis ax = gca()
```

```
# set view to tight, and maximum number of tick intervals to 10
```

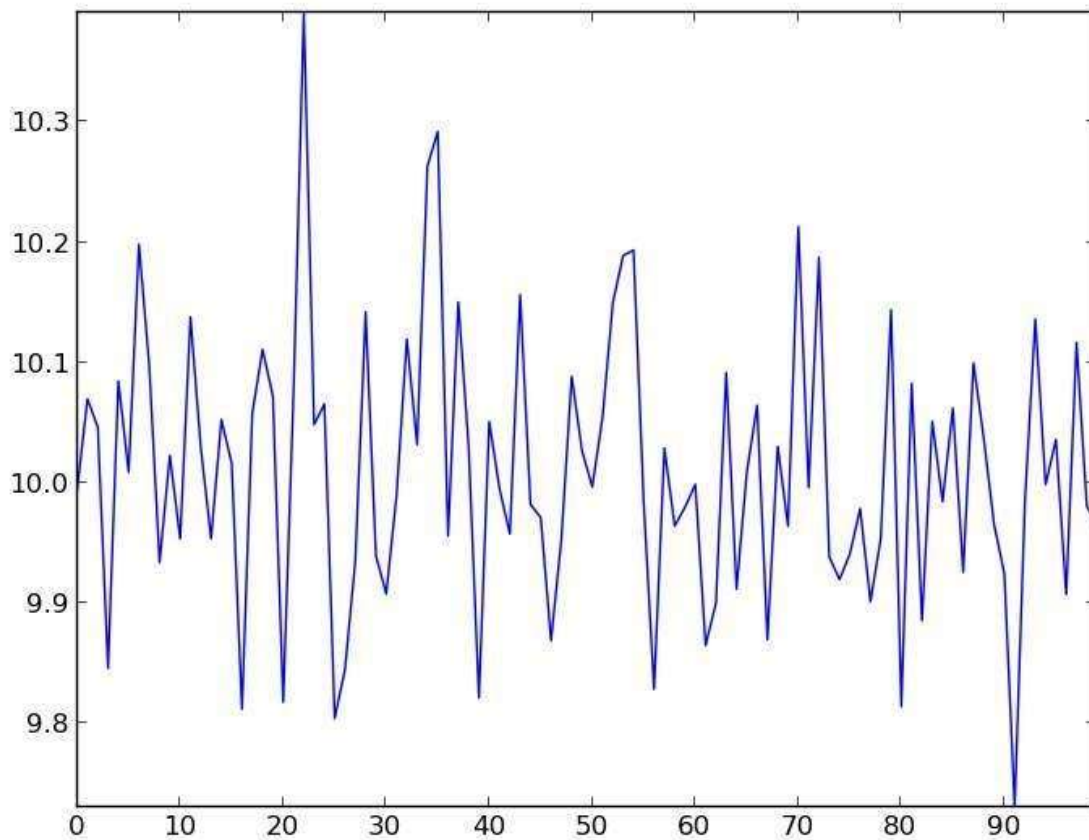
```
ax.locator_params(tight=True, nbins = 10)
```

```
# generate 100 normal distribution values ax.plot(np.random.normal(10, .1,
```

```
100))
```

```
show()
```

This should give us the following graph:



We see how the x and y axes are divided and what values are shown. We could have achieved the same setup using locator classes. Here we are saying "set the major locator to be a multiple of 10":

```
ax.xaxis.set_major_locator(matplotlib.ticker.MultipleLocator(10))
```

Tick formatters can similarly be specified. Formatters specify how the values (usually numbers) are displayed. For example, `matplotlib.ticker.FormatStrFormatter` simply specifies `'%2.1f'` or `'%1.1f cm'` as the string to be used as the label for the ticker.

Let's take a look at one example using dates.

`matplotlib` represents dates in floating point values as the time in days passed since 0001-01-01 UTC plus 1. So, 0001-01-01 UTC 06:00 is 1.25.

Then we can use helper functions such as `matplotlib.dates.date2num()`, `matplotlib.dates.num2date()`, and `matplotlib.dates.drange()` to convert dates between different representations.

Let's see another example:

```
from pylab import *
import matplotlib as mpl import datetime

fig = figure()
# get current axis ax = gca()

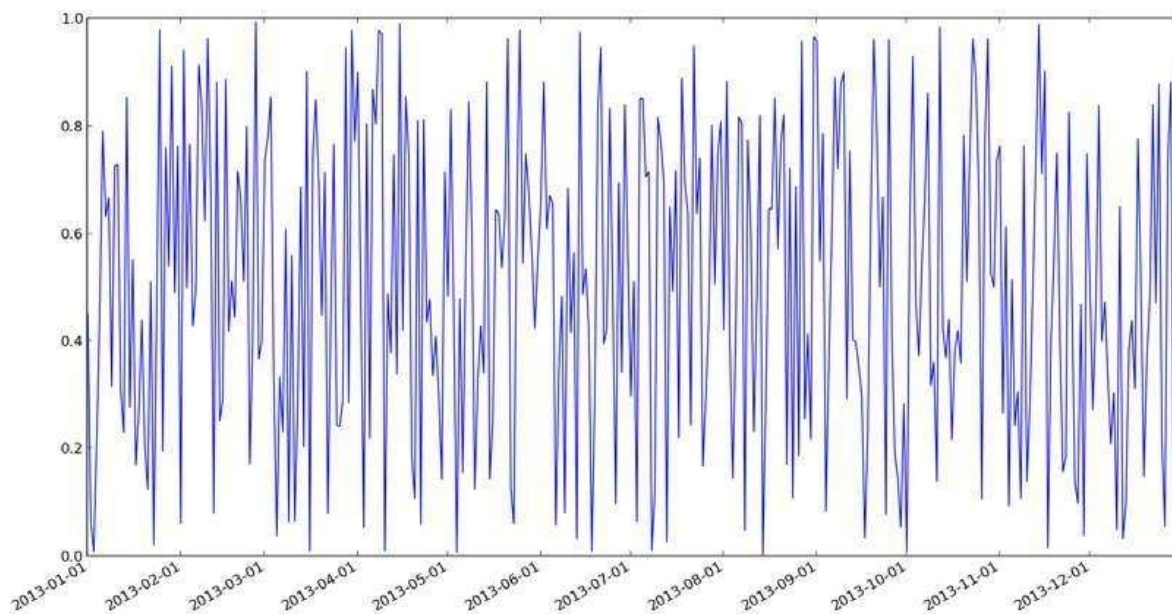
# set some daterange
start = datetime.datetime(2013, 01, 01) stop = datetime.datetime(2013, 12, 31)
delta = datetime.timedelta(days = 1)

# convert dates for matplotlib
dates = mpl.dates.drange(start, stop, delta)
# generate some random values values = np.random.rand(len(dates))
ax = gca()
# create plot with dates
ax.plot_date(dates, values, linestyle='-', marker='')
# specify formater
date_format = mpl.dates.DateFormatter('%Y-%m-%d')
# apply formater
ax.xaxis.set_major_formatter(date_format)

# autoformat date labels
# rotates labels by 30 degrees by default
# use rotate param to specify different rotation degree # use bottom param to
give more room to date labels fig.autofmt_xdate()

show()
```

The preceding code will give us the following graph:



Adding a legend and annotations

Legends and annotations explain data plots clearly and in context. By assigning each plot a short description about what data it represents, we are enabling an easier mental model in the reader's (viewer's) head. This recipe will show how to annotate specific points on our figures and how to create and position data legends.

Getting ready

How many times have you looked at a chart and wondered what the data represents? More often than not, newspapers and other daily and weekly publications create plots that don't contain appropriate legends, thus leaving the reader free to interpret the representation. This creates ambiguity for the readers and increases the possibility of error.

How to do it...

Let us demonstrate how to add legends and annotations with the following example:

```
from matplotlib.pyplot import *
```

```
# generate different normal distributions  
x1 = np.random.normal(30, 3, 100)
```

```

x2 = np.random.normal(20, 2, 100) x3 = np.random.normal(10, 3, 100)

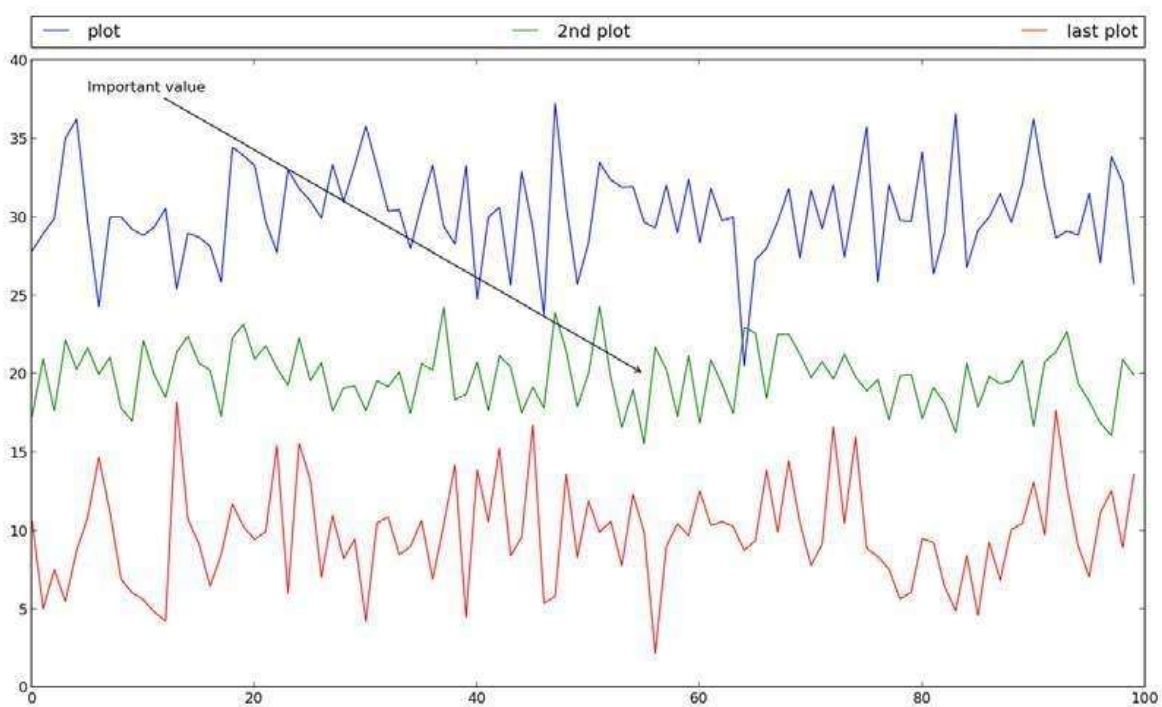
# plot them
plot(x1, label='plot') plot(x2, label='2nd plot') plot(x3, label='last plot')

# generate a legend box
legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=3, mode="expand",
borderaxespad=0.)

# annotate an important value
annotate("Important value", (55,20), xycoords='data', xytext=(5, 38),
arrowprops=dict(arrowstyle='->'))
show()

```

The preceding code will give us the following plot:



What we do is assign a string label with every plot so `legend()` will try and determine what to add in the legend box.

We set the location of a legend box by defining the `loc` parameter. This is optional, but we want to specify a location where it is least likely for the legend box to be drawn over plot lines.

How it works...

All location parameter strings are given in the following table:

String	Number	value	upper right	1
		upper left		2
		lower left		3
		lower right		4
		right		5
		center left		6
		center right		7
		lower center		8
		upper center		9
		center		10

To not show the label in a legend, set the label to `_nolegend_`.

For the legend, we defined the number of columns with `ncol = 3` and set the location with lower left. We specified a bounding box (`bbox_to_anchor`) to start from position (0., 1.02), and to have a width of 1 and a height of 0.102. These are normalized axis coordinates. Parameter mode is either None or expand to allow the legend box to expand horizontally filling the axis area. The parameter `borderaxespad` defines the padding between the axes and the legend border.

For annotations, we have defined a string to be drawn on a plot, on a coordinate `xy`. The coordinate system is specified to be the same as the data one; therefore, coordinate system is `xycoord = 'data'`. The starting position for the text is defined by the value of `xytext`.

An arrow is drawn from `xytext` to `xy` coordinate and the `arrowprops` dictionary can define many properties for that arrow. For this example, we used `arrowstyle` to define arrow style.

Moving spines to the center

This recipe will demonstrate how to move spines to the center.

Spines define data area boundaries; they connect the axis tick marks. There are four spines. We can place them wherever we want; by default, they are placed on the border of the axis, hence we see a box around our data plot.

How to do it...

To move the spines to the center of the plot, we need to remove two spines, making them hidden (set color to none). After that we move two others to coordinate (0,0). The coordinates are specified in data space coordinates.

The following code shows how to do this:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi, np.pi, 500, endpoint=True) y = np.sin(x)
plt.plot(x, y)
ax = plt.gca()

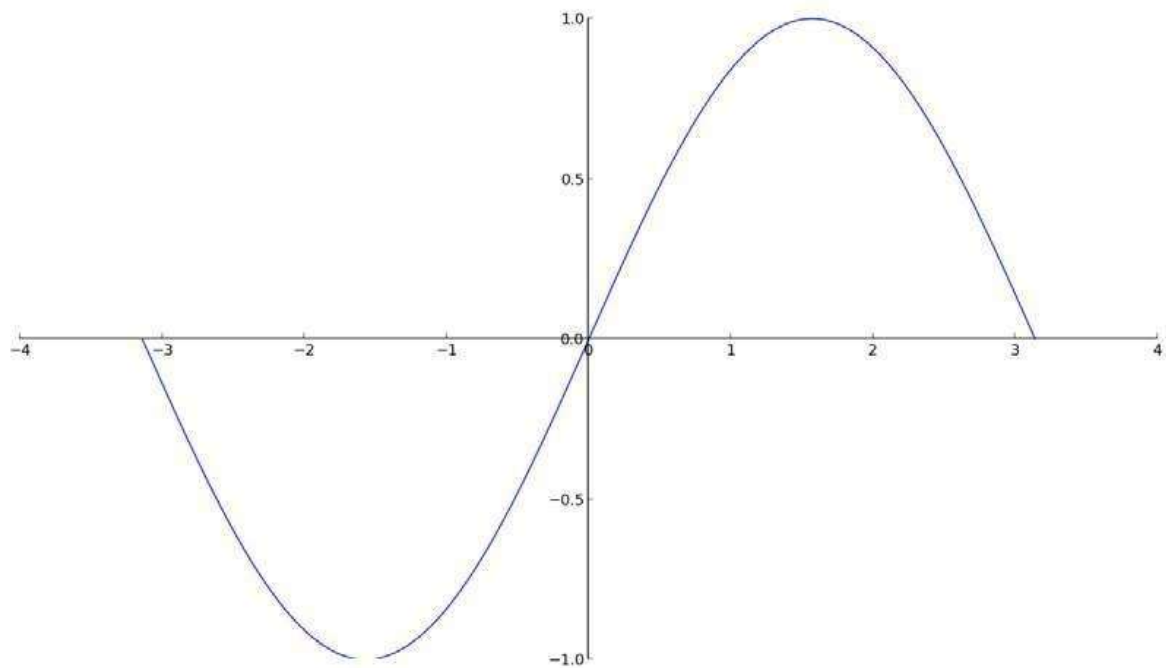
# hide two spines
ax.spines['right'].set_color('none') ax.spines['top'].set_color('none')

# move bottom and left spine to 0,0
ax.spines['bottom'].set_position(('data',0)) ax.spines['left'].set_position(('data',0))

# move ticks positions
ax.xaxis.set_ticks_position('bottom') ax.yaxis.set_ticks_position('left')

plt.show()
```

This is what the plot will look like:



How it works...

This code is dependent on the plot that is drawn because we are moving spines to the location $(0, 0)$ and are plotting a sine function on the interval where $(0, 0)$ is in the middle of the plot. Nevertheless, this demonstrated how to move spines to a particular location and how to get rid of spines we don't want to show.

There's more...

Furthermore, spines can be limited to end where the data ends, for example, using a `set_smart_bounds(True)` call. In this case matplotlib tries to set bounds in a sophisticated way, for example, to handle inverted limits, or to clip line to view if data extends past view.

Making histograms

Histograms are simple, yet it's important to get the right data into them. We will cover histograms in 2D for now.

Histograms are used to visualize estimations of distribution of data. Generally, we use a few terms when speaking of histograms. Vertical rectangles represent frequencies of data points within a particular interval, called a bin. Bins are created at fixed intervals so the total area of a histogram equals the number of data points.

Instead of using absolute values of data, histograms can display relative frequencies of data. When this is the case, the total area equals 1.

Histograms are often used in image manipulation software as a way to visualize image properties, such as distribution of light in a particular color channel. Further, these image histograms can be used in computer vision algorithms to detect peaks aiding in edge detections, image segmentation, and so on.

In *Chapter 5, Making 3D Visualizations*, we have recipes that deal with 3D histograms.

Getting ready

The number of bins is the value we want to get right, but it is hard to get right as there are no strict rules on what is the optimal number of bins. There are different theories on how to calculate number of bins, the simplest being the one based on a ceiling function, where the number of bins (k) is equal to the ceiling $(\max(x) - \min(x)/h)$ where x is the dataset plotted, and h is the desired bin width. This is just one option as the number of bins required to display data properly is dependent on real data distribution.

How to do it...

We create a histogram calling `matplotlib.pyplot.hist()` with a set of parameters. Here are some of the most useful ones:

f bins: This is either an integer number of bins or a sequence giving the bins.

The default is 10.

f range: This is the range of bins and is not used if bins are given as a sequence. Outliers are ignored and the default is None.

f normed: If the value for this is True, histogram values are normalized and form probability density. The default is False.

f histtype: This is the default bar-type histogram. The other options are: ♦

barstacked: This for multiple data gives stacked-view histograms. ♦ **step:** This creates a line plot that is left unfilled.

♦ **stepfilled:** This creates line plot that is filled by default. The default is bar.

f align: This centers bars between bin edges. The default is mid. Other values are left and right.

f color: This specifies the color of the histogram. It may be a single value or have a sequence of colors. If multiple datasets are specified, the color sequence

will be used in the same order. If not specified, a default line color sequence is used.

`f orientation`: This allows the creation of histograms that are horizontal by setting `orientation` to `horizontal`. The default is vertical.

The following code demonstrates how `hist()` is used:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
mu = 100
```

```
sigma = 15
```

```
x = np.random.normal(mu, sigma, 10000)
```

```
ax = plt.gca()
```

```
# the histogram of the data
```

```
ax.hist(x, bins=35, color='r')
```

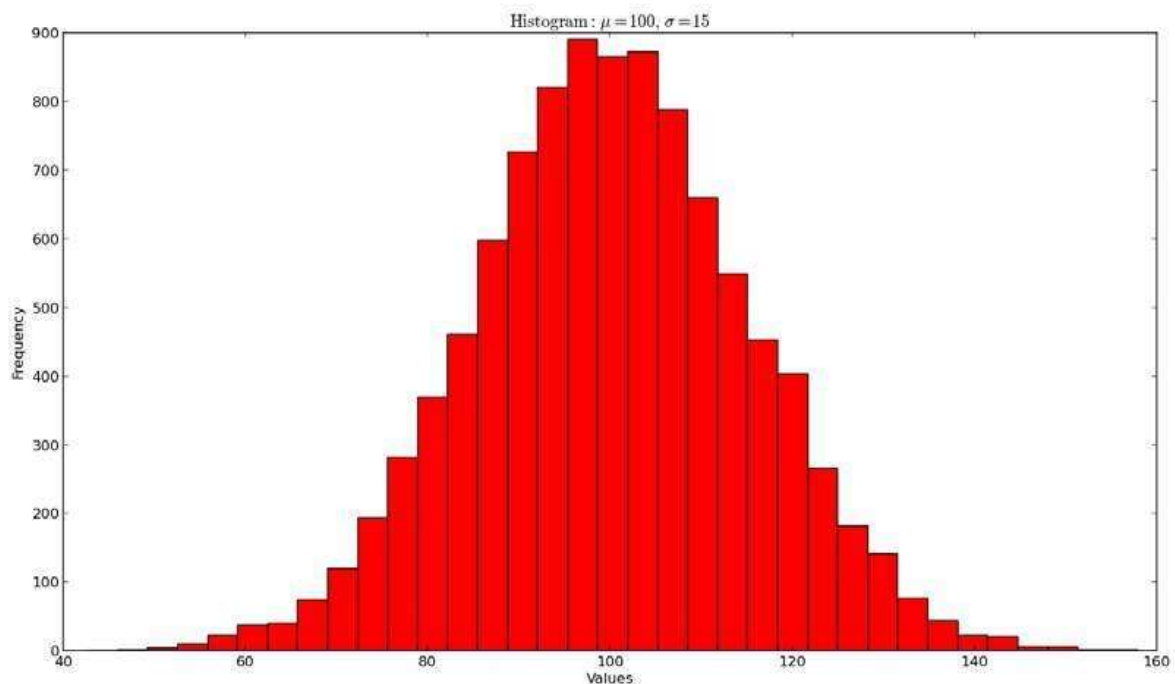
```
ax.set_xlabel('Values')
```

```
ax.set_ylabel('Frequency')
```

```
ax.set_title(r'$\mathrm{Histogram: } \mu = %d, \sigma = %d$' % (mu, sigma))
```

```
plt.show()
```

This creates a neat, red-colored histogram for our data sample:



How it works...

We start by generating some normally distributed data. The histogram is plotted with the specified number of bins—35—and it is normalized by setting `normed` to `True` (or 1); we set the color to `red`.

After that, we set labels and a title for the plot. Here we used the ability to write LaTeX expressions to write math symbols and mixed that with Python format strings.

Making bar charts with error bars

In this recipe, we will show how to create bar charts and how to draw error bars. *Getting ready*

To visualize uncertainty of measurement in our dataset or to indicate the error, we can use error bars. Error bars can easily give an idea of how error free the dataset is. They can show one standard deviation, one standard error, or 95 percent confidence interval. There is no standard here, so always explicitly state what values (errors) error bars display. Most papers in the experimental sciences should contain error bars to present accuracy of the data.

How to do it...

Even though just two parameters are mandatory—left and height—we often want to use more than that. Here are some parameters we can use:

`width`: This gives the width of the bars. The default value is 0.8.

`bottom`: If `bottom` is specified, the value is added to the height. The default is `None`. `edgecolor`: This gives the color of the bar edges.

`ecolor`: This specifies the color of any error bar.

`linewidth`: This gives width of bar edges; special values are `None` (use defaults) and 0 (when bar edges are not displayed).

`orientation`: This has two values `vertical` and `horizontal`.

`xerr` and `yerr`: These are used to generate error bars on the bar chart. Some optional arguments (`color`, `edgecolor`, `linewidth`, `xerr`, and `yerr`) can be single values or sequences with the same length as the number of bars.

How it works...

Let us illustrate this using an example:

```

import numpy as np
import matplotlib.pyplot as plt
# generate number of measurements
x = np.arange(0, 10, 1)
# values computed from "measured"
y = np.log(x)
# add some error samples from standard normal distribution xe = 0.1 *
np.abs(np.random.randn(len(y)))

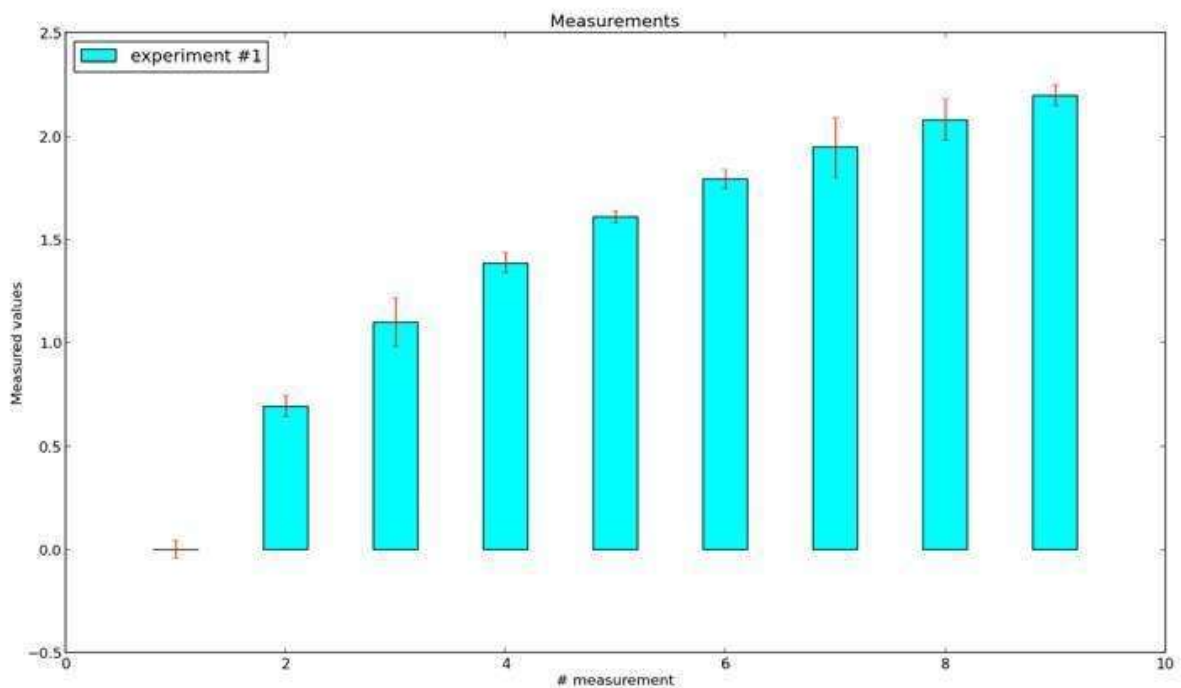
# draw and show errorbar
plt.bar(x, y, yerr=xe, width=0.4, align='center', ecolor='r', color='cyan',
label='experiment #1');

# give some explanations
plt.xlabel('# measurement')
plt.ylabel('Measured values')
plt.title('Measurements')
plt.legend(loc='upper left')

plt.show()

```

The preceding code will plot the following:



To be able to plot an error bar, we needed to have some measures (x); for every measure computed (y), we introduced errors (xe).

We used NumPy to generate and compute values; standard distributions are good enough for demonstration purposes, but if you happen to know your data distribution in advance, you can always make some prototype visualizations and try out different layouts to find the best options to present information.

Another interesting option to use if we are preparing visualizations for a black-and-white medium is hatch; it can have the following values:

Hatch value /

\

|

-

+

x

o

O

.

*

Description

Diagonal hatching Back diagonal

Vertical hatching Horizontal

Crossed

Crossed diagonal Small circle

Large circle

Dot pattern

Star pattern

There's more...

What we have just used are error bars known as symmetrical error bars. If the nature of our dataset is such that errors are not the same in both directions (negative and positive), we can also specify them separately using asymmetrical error bars.

All we have to do differently is to specify xerr or yerr using a two-element list (such as a 2D array), where the first list contains values for negative errors and

the second one for positive errors.

Making pie charts count

Pie charts are special in many ways, the most important being that the dataset they display must sum up to 100 percent or they are just plain not valid.

Getting ready

Pie charts represent numerical proportions, where the arc length of each segment is proportional to the quantity it represents.

They are compact, can look very aesthetically pleasing, but they have been criticized as they can be hard to compare. Another property of pie charts that does not work in their best interest is that pie charts are presented in a specific angle (perspective)—and segments use certain colors—that can skew our perception and influence our conclusion about information presented.

What we will show here is different ways to use pie charts to present data.

How to do it...

As a start, we create a so-called exploded pie chart:

```
from pylab import *
```

```
# make a square figure and axes figure(1, figsize=(6,6))
```

```
ax = axes([0.1, 0.1, 0.8, 0.8])
```

```
# the slices will be ordered
```

```
# and plotted counter-clockwise.
```

```
labels = 'Spring', 'Summer', 'Autumn', 'Winter'
```

```
# fractions are either x/sum(x) or x if sum(x) <= 1 x = [15, 30, 45, 10]
```

```
# explode must be len(x) sequence or None explode=(0.1, 0.1, 0.1, 0.1)
```

```
pie(x, explode=explode, labels=labels, autopct='%1.1f%%', startangle=67)
```

```
title('Rainy days by season')
```

```
show()
```

Pie charts look best if they are inside a square figure and have square axes.

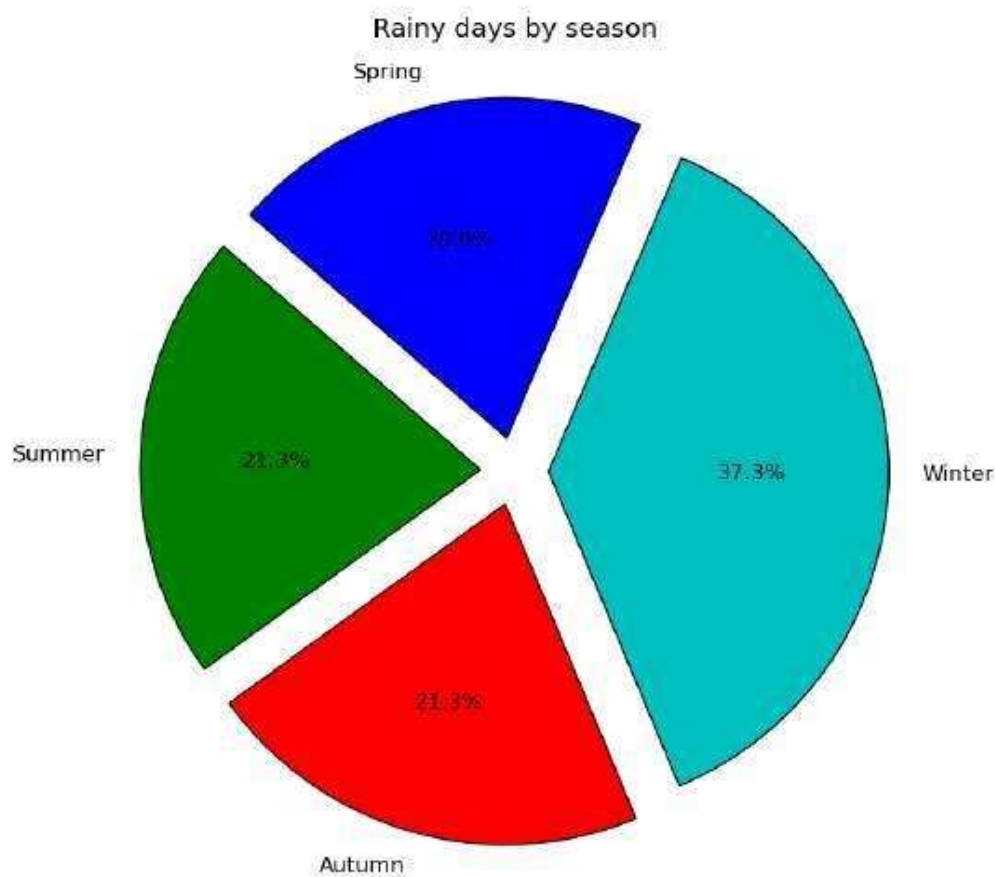
Fractions of the whole sum of the pie chart are defined as $x/\text{sum}(x)$, or x if $\text{sum}(x) \leq 1$. We get the explode effect by defining an explode sequence where each item represents the fraction of radius with which to offset each arc. We use the autopct parameter to format the labels that will be drawn inside the arcs; they

can be a format string or a callable (function).

We can also use a Boolean shadow parameter to add a shadow effect to a pie chart.

If we don't specify startangle, the fractions will be ordered starting counterclockwise from the x axis (angle 0). If we specify 90 as the value of startangle, that will start the pie chart from the y axis.

This is the resulting pie chart:



Plotting with filled areas

In this recipe, we will show you how to fill the area under a curve or in between two different curves.

Getting ready

Library matplotlib allows us to fill areas in between and under the curves with color so that we can display the value of that area to the spectator. Sometimes, it is necessary for readers (viewers) to comprehend the given specialization.

How to do it...

Here's one example of how to fill areas between two contours:

```
from matplotlib.pyplot import figure, show, gca
import numpy as np
x = np.arange(0.0, 2, 0.01)

# two different signals are measured
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = figure()
ax = gca()

# plot and
# fill between y1 and y2 where a logical condition is met ax.plot(x, y1, x, y2,
color='black')

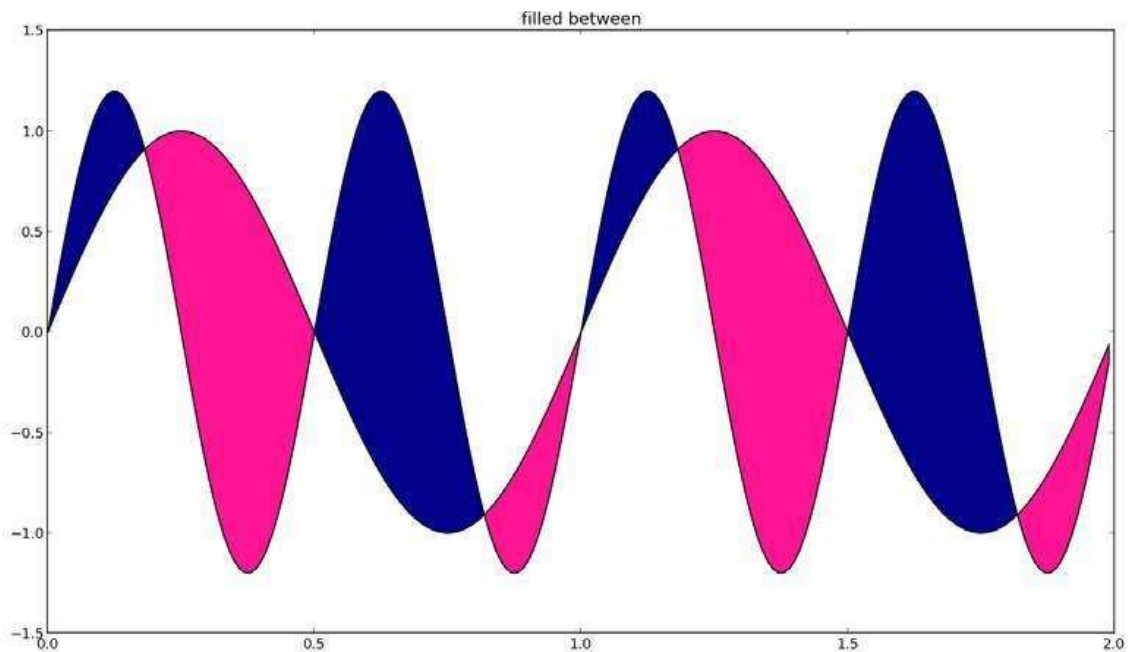
ax.fill_between(x, y1, y2, where=y2>=y1, facecolor='darkblue',
interpolate=True)
ax.fill_between(x, y1, y2, where=y2<=y1, facecolor='deeppink',
interpolate=True)

ax.set_title('filled between')
show()
```

How it works...

After we have generated random signals for a predefined interval, we plot these two signals using a regular plot(). Then we call fill_between() with properties that are required and mandatory.

The function fill_between() is using x as the location from where to pick y values (y1, y2) and will then plot the polygon in certain defined colors. We specify a condition to fill the curve with the where parameter, which accepts Boolean values (can be expressions) so that the fill happens only when the where condition is met.



There's more...

Similar to other functions for plotting, this function also accepts many more parameters, for example, hatch (to specify patterns to fill with instead of color), and line options (linewidth and linestyle).

There is also `fill_betweenx()`, which enables similar fill features, but it does so between horizontal curves.

The more general function `fill()` provides the ability to fill any polygon with a color or a hatch.

Drawing scatter plots with colored markers

If you have two variables and want to spot the correlation between those, a scatter plot may be the solution to spot patterns.

This type of plot is also very usable as a start for more advanced visualization of multidimensional data, for example, to plot a scatter plot matrix.

Getting ready

Scatter plots display values for two sets of data. The data visualization is done as a collection of points not connected by lines. Each of them has its coordinates determined by the value of the variables. One variable is controlled (independent variable) while the other variable is measured (dependent variable) and is often

plotted on the y axis.

How to do it...

Here's a code sample that plots two plots: one with uncorrelated data and other with strong positive correlation:

```
import matplotlib.pyplot as plt
import numpy as np
# generate x values
x = np.random.randn(1000)
# random measurements, no correlation y1 = np.random.randn(len(x))
# strong correlation y2 = 1.2 + np.exp(x)

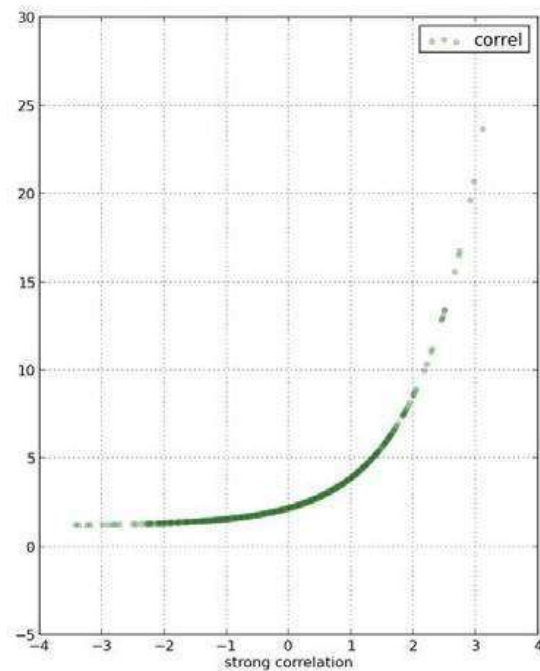
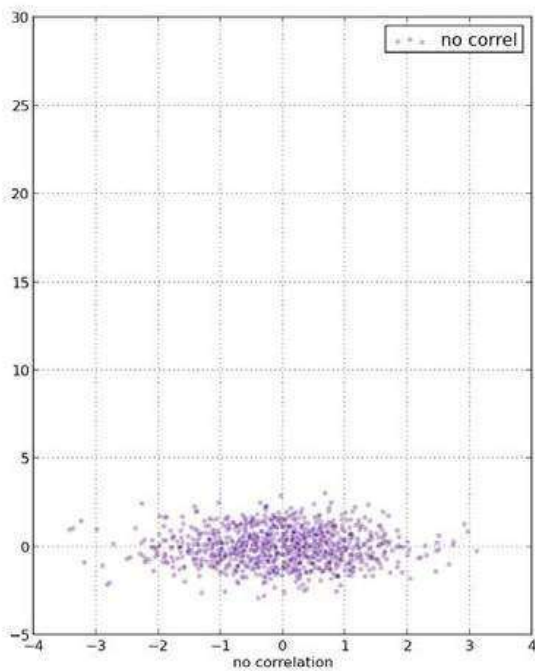
ax1 = plt.subplot(121)
plt.scatter(x, y1, color='indigo', alpha=0.3, edgecolors='white', label='no correl')
plt.xlabel('no correlation')
plt.grid(True)
plt.legend()

ax2 = plt.subplot(122, sharey=ax1, sharex=ax1)
plt.scatter(x, y2, color='green', alpha=0.3, edgecolors='grey', label='correl')
plt.xlabel('strong correlation')
plt.grid(True)
plt.legend()

plt.show()
```

Here, we also use more parameters, such as color for setting the color of the plot, marker for using as a point marker (the default is circle), alpha (alpha transparency), edgecolors (color of the marker edge), and label (for legend box).

These are the plots we get:



How it works...

A scatter plot is often used to identify potential association between two variables, and it's often drawn before working on a fitting regression function. It gives a good visual picture of the correlation, particularly for nonlinear relationships. matplotlib provides the `scatter()` function to plot x versus y unidimensional array of the same length as a scatter plot.

4

More Plots and Customizations

In this chapter we will learn about:

- f Setting the transparency and size of axis labels
- f Adding a shadow to the chart line
- f Adding a data table to the figure
- f Using subplots
- f Customizing grids
- f Creating contour plots
- f Filling an under-plot area
- f Drawing polar plots
- f Visualizing the filesystem tree using a polar bar

Introduction

In this chapter we will explore more advanced properties of the matplotlib library. We are going to introduce more options and will look at how to achieve certain visually pleasing results.

During this chapter we will seek the solutions to some non-trivial problems with representing data when simple charts are not enough. We will try to use more than one type of graph or create hybrid ones to cover some advanced data structures and the representation required.

Setting the transparency and size of axis labels

The Axes label describes what the data in the figure represents and is quite important in the viewer's understanding of the figure itself. By providing labels to the axes background, we help the viewer comprehend the information in an appropriate way.

Getting ready

Before we dive into the code, it is important to understand how matplotlib

organizes our figures.

At the top level, there is a Figure instance containing all that we see and some more (that we don't see). The figure contains, among other things, instances of the Axes class as a field Figure.axes. The Axes instances contain almost everything we care about: all the lines, points, and ticks and labels. So, when we call plot(), we are adding a line (matplotlib.lines.Line2D) to the Axes.lines list. If we plot a histogram (hist()), we are adding rectangles to the list of Axes.patches ("patches" is the term inherited from MATLAB™, and represents the "patch of color" concept).

An instance of Axes also holds references to the XAxis and YAxis instances, which in turn refer to the x axis and y axis, respectively. XAxis and YAxis manage the drawing of the axis, labels, ticks, tick labels, locators, and formatters. We can reference those through Axes.xaxis and Axes.yaxis, respectively. We don't have to go all the way down to XAxis or YAxis instances to get to the labels as matplotlib gives us a helper method (practically a shortcut) that enables iterations via these labels: matplotlib.pyplot.xlabel() and matplotlib.pyplot.ylabel().

How to do it...

We will now create a new figure where we will:

1. Create a plot with some random generated data.
2. Add the title and axes labels.
3. Add alpha settings.
4. Add shadow effects to the title and axes labels.

```
import matplotlib.pyplot as plt from matplotlib import patheffects import numpy  
as np
```

```
data = np.random.randn(70)
```

```
fontsize = 18 plt.plot(data)
```

```
title = "This is figure title" x_label = "This is x axis label" y_label = "This is y  
axis label"
```

```
title_text_obj = plt.title(title, fontsize=fontsize, verticalalignment='bottom')  
title_text_obj.set_path_effects([patheffects.withSimplePatchShadow()])
```



```
# offset_xy -- set the 'angle' of the shadow # shadow_rgbFace -- set the color of
the shadow # patch_alpha -- setup the transparency of the shadow
```

```
offset_xy = (1, -1) rgbRed = (1.0,0.0,0.0) alpha = 0.8
```

```
# customize shadow properties
```

```
pe = patheffects.withSimplePatchShadow(offset_xy = offset_xy,
shadow_rgbFace = rgbRed,
```

```
patch_alpha = alpha) # apply them to the xaxis and yaxis labels
```

```
xlabel_obj = plt.xlabel(x_label, fontsize=fontsize, alpha=0.5)
```

```
xlabel_obj.set_path_effects([pe])
```

```
ylabel_obj = plt.ylabel(y_label, fontsize=fontsize, alpha=0.5)
```

```
ylabel_obj.set_path_effects([pe])
```

```
plt.show()
```

How it works...

We already know all the familiar imports, parts that generate data, and basic plotting techniques so we will skip that. If you are not able to decipher the first few lines of the example, please refer to *Chapter 2, Knowing Your Data*, and *Chapter 3, Drawing Your First Plots and Customizing Them*, where these concepts are already explained. After we plot the dataset, we are ready to add titles and labels, and customize their appearance.

First, we add the title. Then we define the font size and vertical alignment of the title text to be bottom. The default shadow effect is added to the title if we are using matplotlib. `patheffects.withSimplePatchShadow()` with no parameters. The default values for parameters are: `offset_xy=(2,-2)`, `shadow_rgbFace=None`, and `patch_alpha=0.7`. The other values are center, top, and baseline but we choose bottom as the text will have some shadow. In the next line, we add shadow effect. The path effects are part of the matplotlib module `matplotlib.patheffects` that supports `matplotlib.text.Text` and `matplotlib.patches.Patch`.

We now want to add different settings of the shadow to both the x and y axes. First, we customize the position (offset) of the shadow to the parent object, and then we set the color of the shadow. The color is here represented in triples (3-tuple) of float values between 0.0 and 1.0, for each of the RGB channels. Therefore, our red color is represented as (1.0, 0.0, 0.0) (all red, no green, no

blue).

The transparency (or alpha) is set up as a normalized value, and we also want to set this up here to be different from default.

With all the settings there, we instantiate `matplotlib.path_effects` with `SimplePatchShadow` and hold the reference to it in the variable `pe` to reuse it few lines later.

To be able to apply the shadow effect, we need to get to the label object. This is simple enough because `matplotlib.pyplot.xlabel()` returns a reference to the object (`matplotlib.text.Text`) that we use then to call `set_path_effects([pe])`.

We finally show the plot and feel proud of our work.

There's more...

If you are not satisfied with the effects that `matplotlib.path_effects` currently offers, you can inherit the `matplotlib.path_effects._Base` class and override the `draw_path` method. Take a look at the code and comments on how to do that here:

https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/path_effects.py#L47

Adding a shadow to the chart line

To be able to distinguish one particular plot line in the figure or just to fit in the overall style of the output our figure is in, we sometimes need to add a shadow effect to the chart line (or histogram, for that matter). In this recipe we will be learning how to add a shadow effect to the plot's chart lines.

Getting ready

To add shadows to the lines or rectangles in our charts, we need to use the transformation framework built in matplotlib and located in `matplotlib.transforms`. To understand how it all works, we need to explain what transformations are in matplotlib and how they work.

Transformations know how to convert the given coordinates from their coordinate system into display. They also know how to convert them from display coordinates into their own coordinate system.

The following table summarizes existing coordinate systems and what they represent:

Coordinate system	Transformation object	Description
Data	Axes.transData	Represents the user's data coordinate system.
Axes	Axes.transAxes	Represents the Axes coordinate system, where (0,0) represents the bottom-left end of the axes and (1,1) represents the upper-right end of the axes.
Figure	Figure.transFigure	This is the Figure coordinate system, where (0,0) represents the bottom-left end of the figure and (1,1) represents the upper-right end of the figure.
Display	None	Represents the pixel coordinate system of the user display, where (0,0) represents the bottom-left of the display, and tuple (width, height) represents the upper-right of the display, where width and height are in pixels.

Note how the display does not have a value in the column. This is because the default coordinate system is Display, so coordinates are always in pixels relative to your display coordinate systems. This is not very useful, and most often we want them normalized into Figure or Axes or a Data coordinate system. This framework enables us to transform the current object into an offset object, that is, to place that object shifted for a certain distance from the original object.

We will use this framework to create our desired effect on the plotted sine wave.
How to do it...

Here is the code recipe to add shadow to the plotted chart. The code is explained in the section that follows.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms

def setup(layout=None):
```

assert layout is not None

```
fig = plt.figure()
```

```
ax = fig.add_subplot(layout) return fig, ax
```

```
def get_signal():
```

```
t = np.arange(0., 2.5, 0.01) s = np.sin(5 * np.pi * t) return t, s
```

```
def plot_signal(t, s):
```

```
line, = axes.plot(t, s, linewidth=5, color='magenta') return line,
```

```
def make_shadow(fig, axes, line, t, s):
```

```
delta = 2 / 72. # how many points to move the shadow offset =  
transforms.ScaledTranslation(delta, -delta, fig.dpi_
```

```
scale_trans)
```

```
offset_transform = axes.transData + offset
```

```
# We plot the same data, but now using offset transform # zorder -- to render it  
below the line
```

```
axes.plot(t, s, linewidth=5, color='gray',
```

```
transform=offset_transform,
```

```
zorder=0.5 * line.get_zorder())
```

```
if __name__ == "__main__": fig, axes = setup(111) t, s = get_signal()
```

```
line, = plot_signal(t, s)
```

```
make_shadow(fig, axes, line, t, s)
```

```
axes.set_title('Shadow effect using an offset transform') plt.show()
```

How it works...

We start reading the code from the bottom, after the if `__name__` check. First, we create the figure and axes in `setup()`; after that, we obtain a signal (or generate data—sine wave). We plot the basic signal in `plot_signal()`. Then, we make the shadow transformation and plot the shadow in `make_shadow()`.

We use the offset effect to create an offset object underneath and just few points away from original object.

The original object is a simple sine wave that we plot using the standard function `plot()`. To add to this offset transformation, matplotlib contains helper transformation— `matplotlib.transforms.ScaledTranslation`.

The values for `dx` and `dy` are defined in points, and as the point is 1/72 inches, we move the offset object 2pt right and 2pt down.

If you want to learn more about how we converted the point to 1/71 inches, read more in this Wikipedia article: http://en.wikipedia.org/wiki/Point_%28typography%29.

We can use `matplotlib.transforms.ScaledTransformation(xtr, ytr, scaletr)`; here, `xtr` and `ytr` are translation offsets and `scaletr` is a transformation callable to scale `xtr` and `ytr` at transformation time and before display. The most common use case for this is transforming from points to display space: for example, to DPI so that the offset always stays at the same place no matter what the actual output—be it the monitor or printed material. The callable we use for this is already built in, and is available at `Figure.dpi_scale_trans`.

We then plot the same data with the applied transformation.

There's more...

Using transforms to add shadows is just one and not the most popular use case of this framework. To be able to do more with transformation framework, you will need to learn the details of how the transformation pipeline works and what the extension points are (what classes to inherit and how). It's easy enough because matplotlib is open source, and even if some code is not well documented, there is a source you can read from and use or change, thus contributing to the overall quality and usefulness of matplotlib.

Adding a data table to the figure

Although matplotlib is mainly a plotting library, it helps us with small errands when we are creating a chart, such as having a neat data table beside our beautiful chart. In this recipe we will be learning how to display a data table alongside the plots in the figure.

Getting ready

It is important to understand why we are adding a table to a chart. The main

intention of plotting data visually is to explain the otherwise not understandable (or hardly understandable) data values. Now, we want to add that data back. It is not wise just to cram a big table with values underneath the chart.

But, carefully picked, maybe the summed or highlighted values from the whole, charted dataset can identify important parts of the chart or emphasize the important values for those places where the exact value (for example, yearly sales in USD) is important (or even required).

How to do it...

Here's the code to add a sample table to our figure:

```
import matplotlib.pyplot as plt
import numpy as np

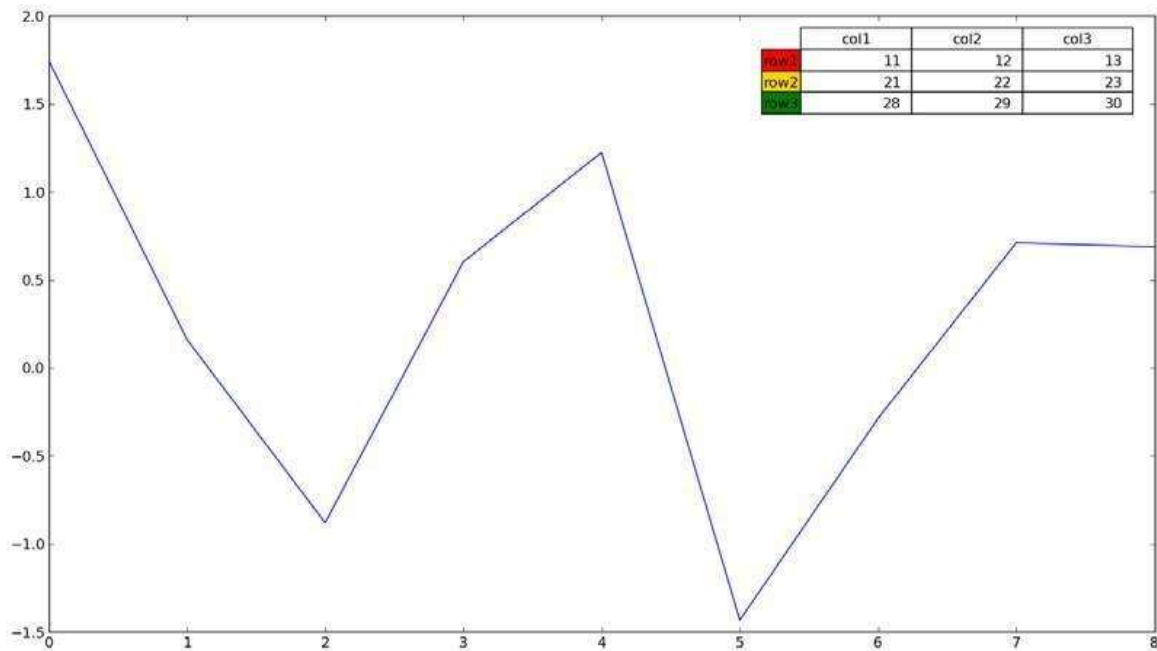
plt.figure()
ax = plt.gca()
y = np.random.randn(9)

col_labels = ['col1','col2','col3']
row_labels = ['row1','row2','row3']
table_vals = [[11, 12, 13], [21, 22, 23], [28, 29, 30]] row_colors = ['red', 'gold',
'green']
my_table = plt.table(cellText=table_vals,

colWidths=[0.1] * 3, rowLabels=row_labels, colLabels=col_labels,
rowColours=row_colors, loc='upper right')

plt.plot(y) plt.show()
```

The previous code snippet gives a plot such as the following:



How it works...

Using `plt.table()` we create a table of cells and add it to the current axes. The table can have (optional) row and column headers. Each table cell contains either patch or text. The column widths and row heights for the table can be specified. The return value is a sequence of objects (text, line, and patch instances) that the table is made of.

The basic function signature is:

```
table(cellText=None, cellColours=None,
      cellLoc='right', colWidths=None,
      rowLabels=None, rowColours=None, rowLoc='left', colLabels=None,
      colColours=None, colLoc='center', loc='bottom', bbox=None)
```

The function instantiates and returns the `matplotlib.table.Table` instance. This is usually the case with matplotlib; there's just one way to add the table to the figure. The object-oriented interface can be directly accessed. We can use the `matplotlib.table.Table` class directly to fine-tune our table before we add it onto our axes instance with `add_table()`.

There's more...

You can have more control if you directly create an instance of

`matplotlib.table.Table` and configure it before you add it to the axes instance. You can add the table instance to axes using `Axes.add_table(table)`, where `table` is an instance of `matplotlib.table.Table`.

Using subplots

If you are reading this book from the start, you are probably familiar with the subplot class, a descendant of axes that lives on the regular grid of subplot instances. We are going to explain and demonstrate how to use subplots in advanced ways.

In this recipe we will be learning how to create custom subplot configurations on our plots.

Getting ready

The base class for subplots is `matplotlib.axes.SubplotBase`. These subplots are `matplotlib.axes.Axes` instances but provides helper methods for generating and manipulating a set of Axes within a figure.

There is a class `matplotlib.figure.SubplotParams`, which holds all the parameters for subplot. The dimensions are normalized to the width or height of the figure. As we already know, if we don't specify any custom values, they will be read from the rc parameters.

The scripting layer (`matplotlib.pyplot`) holds a few helper methods to manipulate subplots.

`matplotlib.pyplot.subplots` is used for the easy creation of common layouts of subplots. We can specify the size of the grid—the number of rows and columns of the subplot grid.

We can create subplots that share the x or y axes. This is achieved using `sharex` or the `sharey` keyword argument. The argument `sharex` can have the value `True`, in which case the x axis is shared among all the subplots. The tick labels will be invisible on all but the last row of plots. They can also be defined as `String`, with enumerated values of `row`, `col`, `all`, or `none`. The value `all` is the same as `True`, and the value `none` is the same as `False`. If the value `row` is specified, each subplot row shares the x axis. If the value `col` is specified, each subplot column shares the x axis. This helper returns tuple `fig, ax` where `ax` is either an axis instance or, if more than one subplot is created, an array of axis instances.

`matplotlib.pyplot.subplots_adjust` is used to tune the subplot layout. The keyword arguments specify the coordinates of the subplots inside the figure (left, right, bottom, and top) normalized to figure size. White space can be specified to be left between the subplots using the `wspace` and `hspace` arguments for width and height amounts respectively.

How to do it...

1. We will show you an example of using yet another helper function in the matplotlib toolkit—`subplot2grid`. We define the grid's geometry and the subplot location. Note that this location is 0-based, not 1-based as we are used to in `plot.subplot()`. We can also use `colspan` and `rowspan` to allow the subplot to span multiple columns and rows in a given grid. For example, we will: create a figure; add various subplot layouts using `subplot2grid`; reconfigure the tick label size.

2. Show the plot:

```
import matplotlib.pyplot as plt
```

```
plt.figure(0)
```

```
axes1 = plt.subplot2grid((3, 3), (0, 0), colspan=3) axes2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)
```

```
axes3 = plt.subplot2grid((3, 3), (1, 2))
```

```
axes4 = plt.subplot2grid((3, 3), (2, 0))
```

```
axes5 = plt.subplot2grid((3, 3), (2, 1), colspan=2)
```

```
# tidy up tick labels size
```

```
all_axes = plt.gcf().axes
```

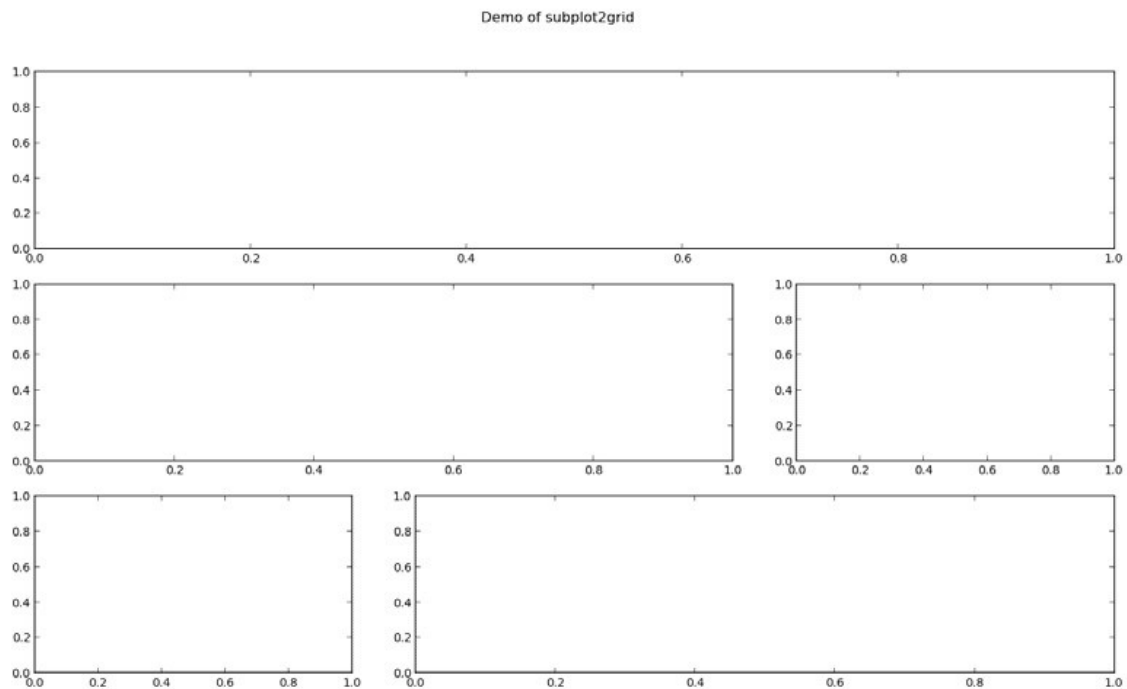
```
for ax in all_axes:
```

```
    for ticklabel in ax.get_xticklabels() + ax.get_yticklabels():
```

```
        ticklabel.set_fontsize(10)
```

```
plt.suptitle("Demo of subplot2grid") plt.show()
```

When we execute the previous code, the following plot is created:



How it works...

We provide `subplot2grid` with a shape, location (`loc`), and optionally, `rowspan` and `colspan`. The important difference here is that the location is indexed from 0, and not from 1, as in `figure.add_subplot`.

There's more...

To give an example of another way you can customize the current axes or subplot:

```
axes = fig.add_subplot(111)
rectangle = axes.patch
rectangle.set_facecolor('blue')
```

Here we see that every axes instance contains a field `patch` referencing the `rectangle` instance, thus representing the background of the current axes instance. This instance has properties that we can update, hence updating the current axes background. We can change its color, but we can also load an image to add a watermark protection, for example. It is also possible to create a patch first and then just add it to the axes background:

```
fig = plt.figure()
```

```
axes = fig.add_subplot(111)
rect = matplotlib.patches.Rectangle((1,1), width=6, height=12)
axes.add_patch(rect)
# we have to manually force a figure draw
axes.figure.canvas.draw()
```

Customizing grids

A grid is usually handy to have under lines and charts as it helps the human eye spot differences in pattern and compare plots visually in the figure. To be able to set up how visibly, how frequently, and in what style the grid is displayed—or whether it is displayed at all—we should use `matplotlib.pyplot.grid`.

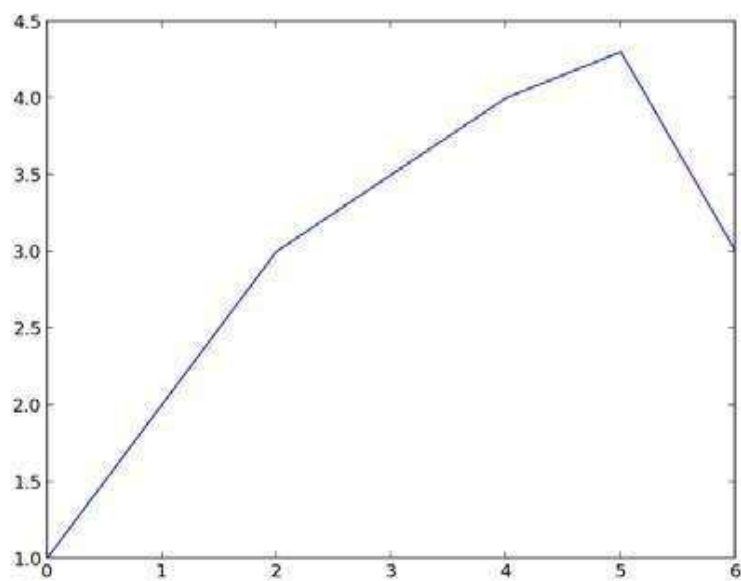
In this recipe we will be learning how to turn the grid on and off, and how to change the major and minor ticks on a grid.

Getting ready

The most frequent grid customization is reachable in the `matplotlib.pyplot.grid` helper function.

To see the interactive effect of this, you should run the following under `ipython – pylab`. The basic call to `plt.grid()` will toggle grid visibility in the current interactive session started by the last IPython PyLab environment:

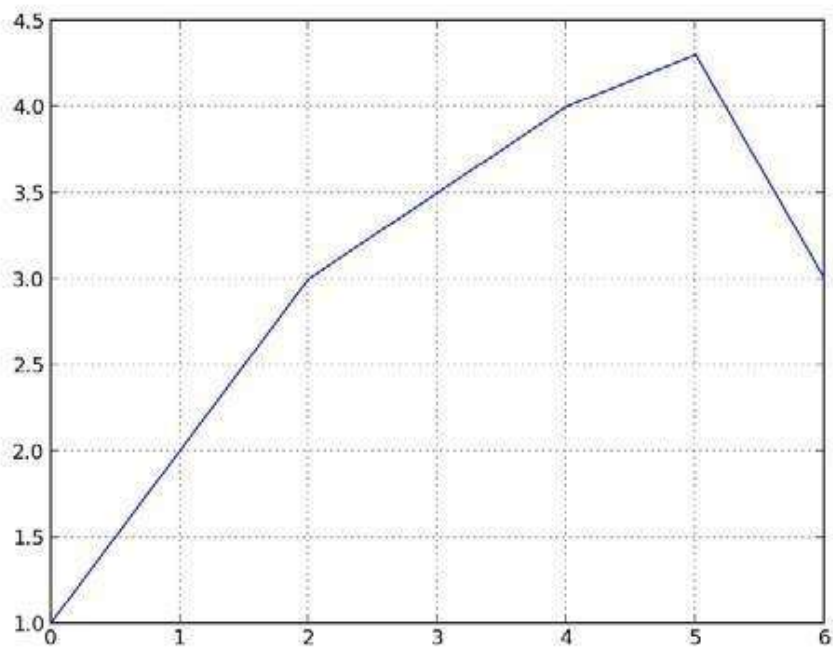
```
In [1]: plt.plot([1,2,3,3.5,4,4.3,3])
Out[1]: [<matplotlib.lines.Line2D at 0x3dcc810>]
```



Now we can toggle the grid on the same figure:

In [2]: `plt.grid()`

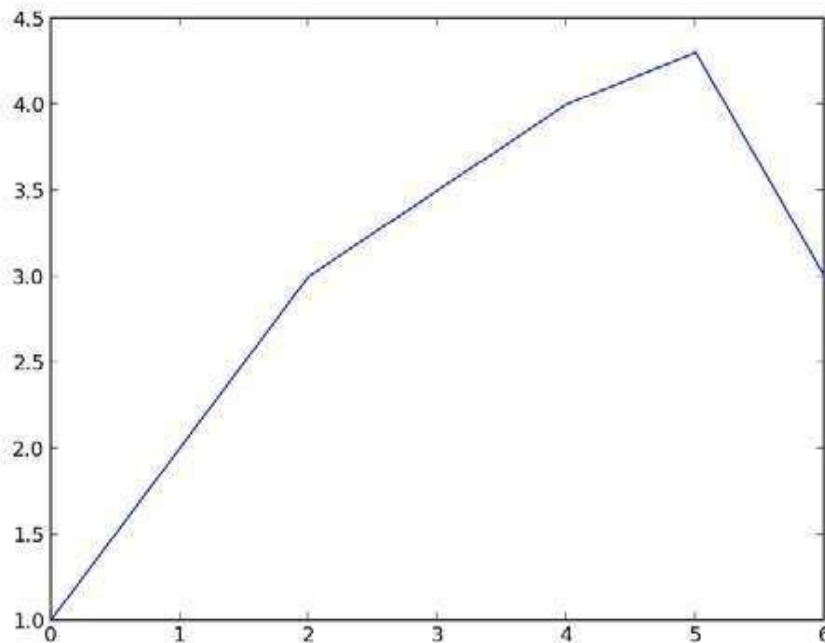
We turn the grid back on, as shown in the following plot:



And then we

turn it off again:

```
In [3]: plt.grid()
```



Apart from just turning them on and off, we can further customize the grid appearance. We can manipulate the grid with just major ticks, or just minor ticks, or both; hence, the value of function argument which can be 'major', 'minor', or 'both'. Similar to this, we can control the horizontal and vertical ticks separately by using the argument axis that can have values 'x', 'y', or 'both'.

All the other properties are passed via kwargs and represent a standard set of properties that a `matplotlib.lines.Line2D` instance can accept, such as color, linestyle, and linewidth; here is an example:

```
ax.grid(color='g', linestyle='--', linewidth=1)
```

How to do it...

This is nice, but we want to be able to customize more. In order to do that we need to reach deeper into matplotlib and into `mpl_toolkits` and find the `AxesGrid` module that allows us to make grids of axes in an easy and manageable way:

```
import numpy as np
import matplotlib.pyplot as plt
```

```

from mpl_toolkits.axes_grid1 import ImageGrid
from matplotlib.cbook import get_sample_data

def get_demo_image():
    f = get_sample_data("axes_grid/bivariate_normal.npy",
        asfileobj=False)
    # z is a numpy array of 15x15
    Z = np.load(f)
    return Z, (-3, 4, -4, 3)

def get_grid(fig=None, layout=None, nrows_ncols=None): assert fig is not None
    assert layout is not None
    assert nrows_ncols is not None

    grid = ImageGrid(fig, layout, nrows_ncols=nrows_ncols, axes_pad=0.05,
        add_all=True, label_mode="L")
    return grid

def load_images_to_grid(grid, Z, *images):
    min, max = Z.min(), Z.max()
    for i, image in enumerate(images):

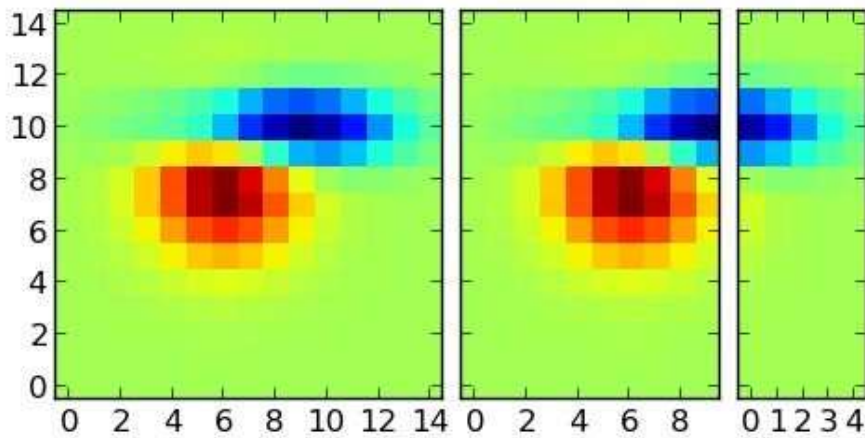
        axes = grid[i]
        axes.imshow(image, origin="lower", vmin=min, vmax=max,
            interpolation="nearest")

if __name__ == "__main__":
    fig = plt.figure(1, (8, 6)) grid = get_grid(fig, 111, (1, 3)) Z, extent =
        get_demo_image()

    # Slice image
    image1 = Z
    image2 = Z[:, :10] image3 = Z[:, 10:]

    load_images_to_grid(grid, Z, image1, image2, image3)
    plt.draw() plt.show()
    The given code will render the following plot:

```



How it works...

In the function `get_demo_image`, we loaded data from the sample data directory that comes with matplotlib.

The list grid holds our axes grid (in this case, `ImageGrid`).

The variables `image1`, `image2`, and `image3` hold sliced data from `Z` that we have split over multiple axes in the list grid.

Looping over all the grids, we are plotting data from `im1`, `im2`, and `im3` using the standard `imshow()` call, while matplotlib takes care that everything is neatly rendered and aligned.

Creating contour plots

A contour plot displays the isolines of matrix. Isolines are curves where a function of two variables has the same value.

In this recipe we will learn how to create contour plots.

Getting ready

Contours are represented as a contour plot of matrix `Z`, where `Z` is interpreted as height with respect to the X-Y plane. `Z` is of minimum size 2 and must contain at least two different values.

The problem with contour plots is that if they are coded without labeling the isolines, they render pretty useless as we cannot decode the high points from low points or find local minimas.

Here we need to label the contour also. The labeling of isolines can be done either by using labels (`clabel()`) or `colormaps`. If your output medium permits the

usage of color, colormaps are preferred because viewers will be able to decode data more easily.

The other risk with contour plots is choosing the number of isolines to plot. If we choose too many, the plot becomes too dense to decode, and if we go with too few isolines, we lose information and can perceive data differently.

The function `contour()` will automatically guess how many isolines to plot, but we also have the ability to specify our own number.

In matplotlib, we draw contour plots using `matplotlib.pyplot.contour`.

There are two similar functions: `contour()` draws contour lines, and `contourf()` draws filled contours. We are going to demonstrate only `contour()`, but almost everything is applicable to `contourf()`. They understand almost the same arguments as well. The function `contour()` can have different call signatures, depending on what data we have and/or what the properties that we want to visualize are.

Call signature

`contour(Z)`

`contour(X,Y,Z)`

`contour(Z,N)`

`contour(X,Y,Z,N)`

`contour(Z,V)`

`contour(X,Y,Z,V) contourf(..., V)`

`contour(Z, **kwargs)`

Description

Plots the contour of `Z` (array). The level values are chosen automatically.

Plots the contour of `X`, `Y`, and `Z`. The arrays `X` and `Y` are (x, y) surface coordinates.

Plots the contour of `Z`, where the number of levels is defined with `N`. The level values are automatically chosen. Plots the contour lines with levels at the values specified in `V`.

Fills the `len(V)-1` regions between the level values in sequence `V`.

Uses keyword arguments to control common line properties (colors, line width, origin, color map, and so on).

There exist certain constraints on the dimensionality and shape of X, Y, and Z. For example, X and Y can be of two dimensions and of the same shape as Z. If they are of one dimension, such that the length of X is equal to the number of columns in Z, then the length of Y will be equal to the number of rows in Z.

How to do it...

In the following code example, we will:

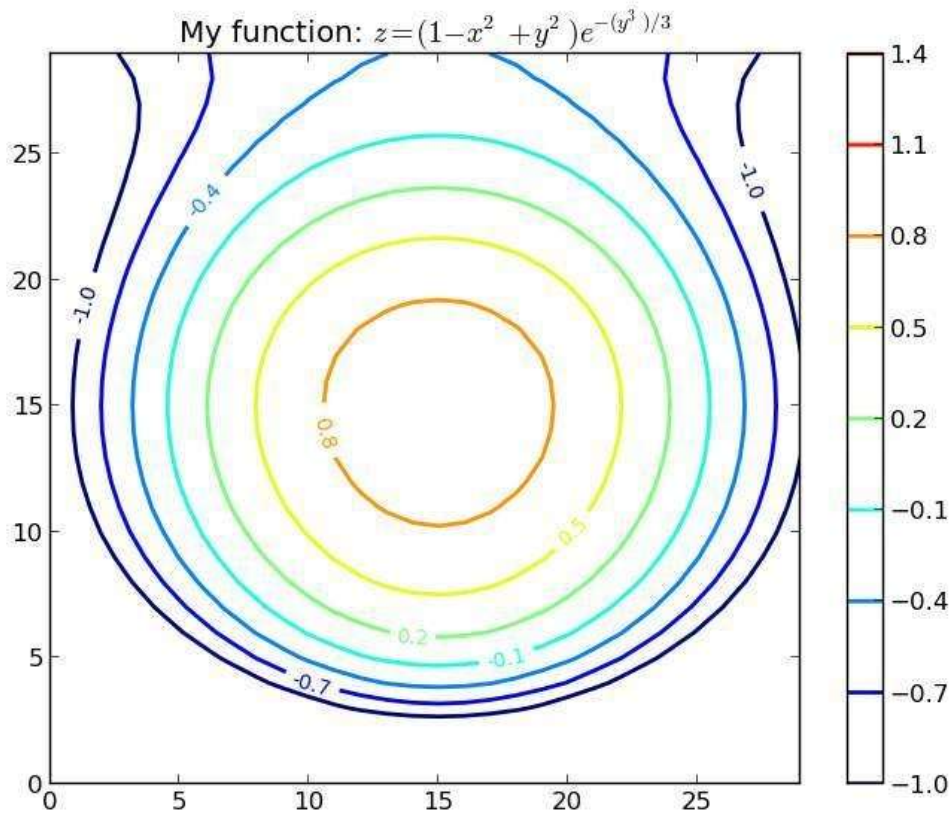
1. Implement a function to act as a mock signal processor.
2. Generate some linear signal data.
3. Transform the data into suitable matrices for use in matrix operations.
4. Plot contour lines.
5. Add contour line labels.
6. Show the plot.
7. Import numpy as np.
8. Import matplotlib as mpl.
9. Import matplotlib.pyplot as plt.

```
def process_signals(x,y):
    return (1 - (x ** 2 + y ** 2)) * np.exp(-y ** 3 / 3)
x = np.arange(-1.5, 1.5, 0.1) y = np.arange(-1.5, 1.5, 0.1)
# Make grids of points X,Y = np.meshgrid(x, y)
Z = process_signals(X, Y)
# Number of isolines
N = np.arange(-1, 1.5, 0.3)

# adding the Contour lines with labels
CS = plt.contour(Z, N, linewidths=2, cmap=mpl.cm.jet) plt.clabel(CS,
inline=True, fmt='%1.1f', fontsize=10) plt.colorbar(CS)

plt.title('My function: $z=(1-x^2+y^2) e^{\{-(y^3)/3\}}$') plt.show()
```

This will give us the following chart:



How it works...

We reached for little helpers from numpy to create our ranges and matrices. After we evaluated my_function into Z, we simply called contour, providing Z and the number of levels for isolines.

At this point, try experimenting with the third parameter in the N arange() call. For example, instead of `N = np.arange(-1, 1.5, 0.3)`, try changing 0.3 to 0.1 or 1 to experience how the same data is seen differently, depending on how we encode the data in a contour plot.

We also added a color map by simply giving it CS (a matplotlib.contour.QuadContourSet instance).

Filling an under-plot area

The basic way to draw a filled polygon in matplotlib is to use `matplotlib.pyplot.fill`. This function accepts similar arguments as `matplotlib.pyplot.plot`—multiple x and y pairs and other Line2D properties. This

function returns the list of Patch instances that were added.

In this recipe we will learn how to shade certain areas of plot intersections.

Getting ready

matplotlib provides several functions to help us plot filled figures, apart from plotting functions that are inherently plotting closed filled polygons, such as `histogram()`, of course.

We already mentioned one—`matplotlib.pyplot.fill`—but there are the `matplotlib.pyplot.fill_between()` and `matplotlib.pyplot.fill_betweenx()` functions too. These functions fill the polygons between two curves. The main difference between `fill_between()` and `fill_betweenx()` is that the latter fills between the x axis values, whereas the former fills between the y axis values.

The function `fill_between` accepts argument `x`—an x axis array of data—and `y1` and `y2`—the y axis arrays of the data. Using arguments, we can specify conditions under which the area will be filled. This condition is the Boolean condition, usually specifying the y axis value ranges. The default value is `None`—meaning, to fill everywhere.

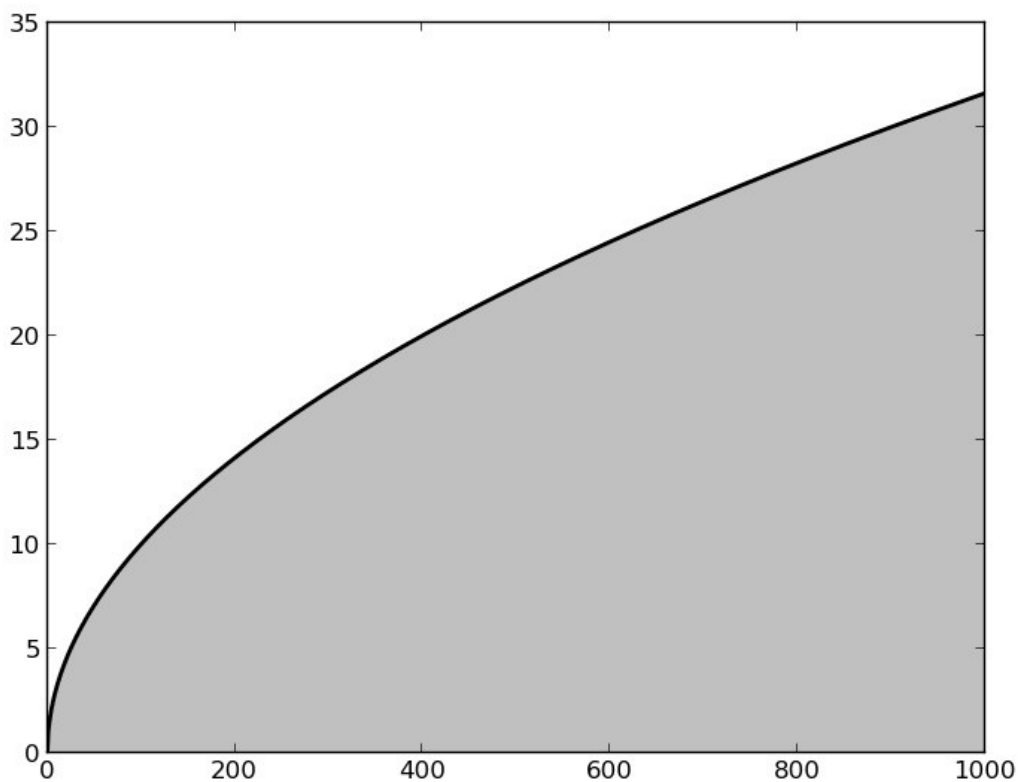
How to do it...

To start off with a simple example, we will fill the area under a simple function:

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

t = range(1000)
y = [sqrt(i) for i in t]
plt.plot(t, y, color='red', lw=2) plt.fill_between(t, y, color='silver') plt.show()
```

The previous code gives us the following plot:



This is fairly straightforward and gives an idea how `fill_between()` works. Note how we needed to plot the actual function line (using `plot()`, of course), where `fill_between()` just draws a polygonal area filled with color ('silver'). We will demonstrate another recipe here. It will involve more conditioning for the fill function. The following is the code for the example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01) y1 = np.sin(np.pi*x)
y2 = 1.7*np.sin(4*np.pi*x)

fig = plt.figure()
axes1 = fig.add_subplot(211)
axes1.plot(x, y1, x, y2, color='grey')
axes1.fill_between(x, y1, y2, where=y2<=y1, facecolor='blue', interpolate=True)
axes1.fill_between(x, y1, y2, where=y2>=y1, facecolor='gold', interpolate=True)
```

```

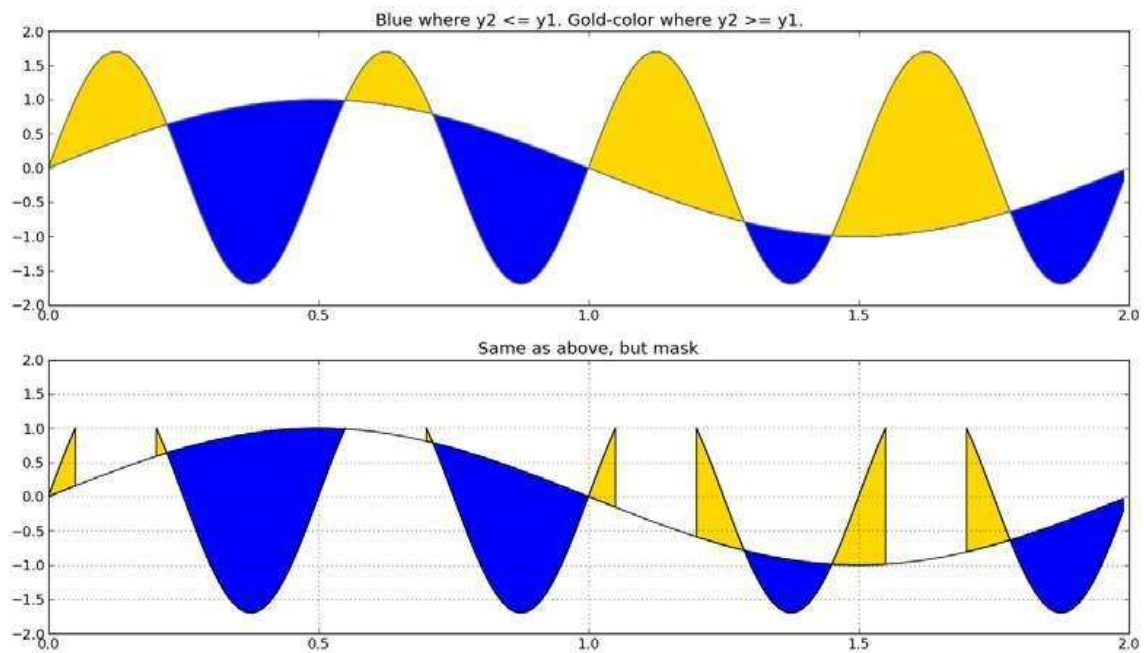
axes1.set_title('Blue where  $y_2 \leq y_1$ . Gold-color where  $y_2 \geq y_1$ .')
axes1.set_ylim(-2,2)

# Mask values in y2 with value greater than 1.0
y2 = np.ma.masked_greater(y2, 1.0)
axes2 = fig.add_subplot(212, sharex=axes1)
axes2.plot(x, y1, x, y2, color='black')
axes2.fill_between(x, y1, y2, where=y2<=y1, facecolor='blue', interpolate=True)
axes2.fill_between(x, y1, y2, where=y2>=y1, facecolor='gold', interpolate=True)
axes2.set_title('Same as above, but mask')
axes2.set_ylim(-2,2)
axes2.grid('on')

plt.show()

```

The preceding code will render the following plot:



How it works...

For this example, we first created two sinusoidal functions that overlap at certain points. We also created two subplots to compare the two variations that render filled regions.

In both cases, we used `fill_between()` with an argument, `where`, that accepts an

N-length Boolean array and will fill over the regions where equals True.

The bottom subplot illustrates `mask_greater`, which masks an array at values greater than a given value. This is a function from the `numpy.ma` package to handle missing or invalid values. We turned the grid on the bottom axes to make it easier to spot this.

Drawing polar plots

If the data is already represented using polar coordinates, we can as well display it using polar figures. Even if the data is not in polar coordinates, we should consider converting it to polar form and draw on polar plots.

To answer whether we want to do this, we need to understand what the data represents and what we are hoping to display to the end user. Imagining what the user will read and decode from our figures leads us usually to the best of visualizations.

Polar plots are commonly used to display information that is radial in nature. For example, in sun path diagrams—we see the sky in radial projection, and the radiation maps of antennas radiate differently at different angles. You can learn more about this at: <http://www.astronwireless.com/topic-archives-antenna-radiation-patterns.asp>.

In this recipe, we will learn how to change the coordinate system used in the plot and to use the polar coordinate system instead.

Getting ready

To display data in polar coordinates, we must have appropriate data values. In the polar coordinate system, a point is described with radius distance (usually denoted with r) and angle (usually *theta*). The angle can be in radians or degrees, but matplotlib uses degrees.

Similar enough to the function `plot()`, to draw polar plots, we will use the function `polar()`, which accepts two same-length arrays of parameters, `theta` and `r`, for angle array and radius array, respectively. The function also accepts other formatting arguments, the same ones as `plot()` one does.

We also need to tell matplotlib that we want axes in the polar coordinate system. This is done by providing the `polar=True` argument to the `add_axes` or

add_subplot functions.

Additionally, to set other properties on the figure, such as grids on radii or angles, we need to use `matplotlib.pyplot.rgrids()` to toggle radial grid visibility or to set up labels. Similarly, we use `matplotlib.pyplot.thetagrid()` to configure angle ticks and labels.

How to do it...

Here is one recipe that demonstrates how to plot polar bars:

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

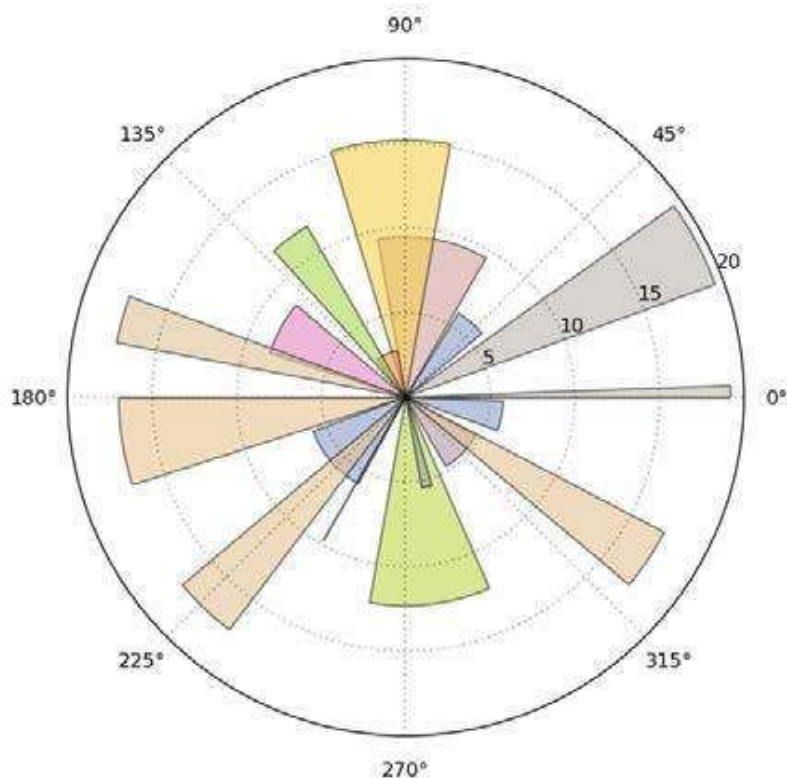
figsize = 7
colormap = lambda r: cm.Set2(r / 20.) N = 18 # number of bars

fig = plt.figure(figsize=(figsize,figsize)) ax = fig.add_axes([0.2, 0.2, 0.7, 0.7],
polar=True)

theta = np.arange(0.0, 2*np.pi, 2*np.pi/N) radii = 20*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = ax.bar(theta, radii, width=width, bottom=0.0) for r, bar in zip(radii, bars):

bar.set_facecolor(colormap(r))
bar.set_alpha(0.6)
plt.show()
```

The preceding code snippet will give us the following plot:



How it works...

First, we create a square figure and add the polar axes to it. The figure does not have to be square, but then our polar plot will be ellipsoidal.

We then generate random values for a set of angles (theta) and a set of polar distances (radii). Because we drew bars, we also needed a set of widths for each bar, so we also generated a set of widths. Since `matplotlib.axes.bar` accepts an array of values (as almost all the drawing functions in `matplotlib` do) we don't have to loop over this generated dataset; we just need to call the bar once with all the arguments passed to it.

In order to make every bar easily distinguishable, we have to loop over each bar added to `ax` (`Axes`) and customize its appearance (face-color and transparency).

Visualizing the filesystem tree using a polar bar

We want to show in this recipe how to solve a "real-world" task—how to use `matplotlib` to visualize our directory occupancy.

In this recipe we will learn how to visualize a filesystem tree with relative sizes.

Getting ready